Juan Nathaniel Follow
May 9, 2021 · 6 min read ★ · Listen

OPINION

Golang for Machine Learning?

Is Golang the future for building a Machine Learning pipeline? Let's try building one.

Go or Golang was designed at Google in 2007 and is syntactically similar to C, but with memory safety, garbage collection, and structural typing. In addition to its blazingly fast performance, Go, unlike Python, allows for easy concurrency just like in C++ or Java. Concurrency allows multiple programs or algorithms (including those of ML) to be executed asynchronously without affecting the final outcome.

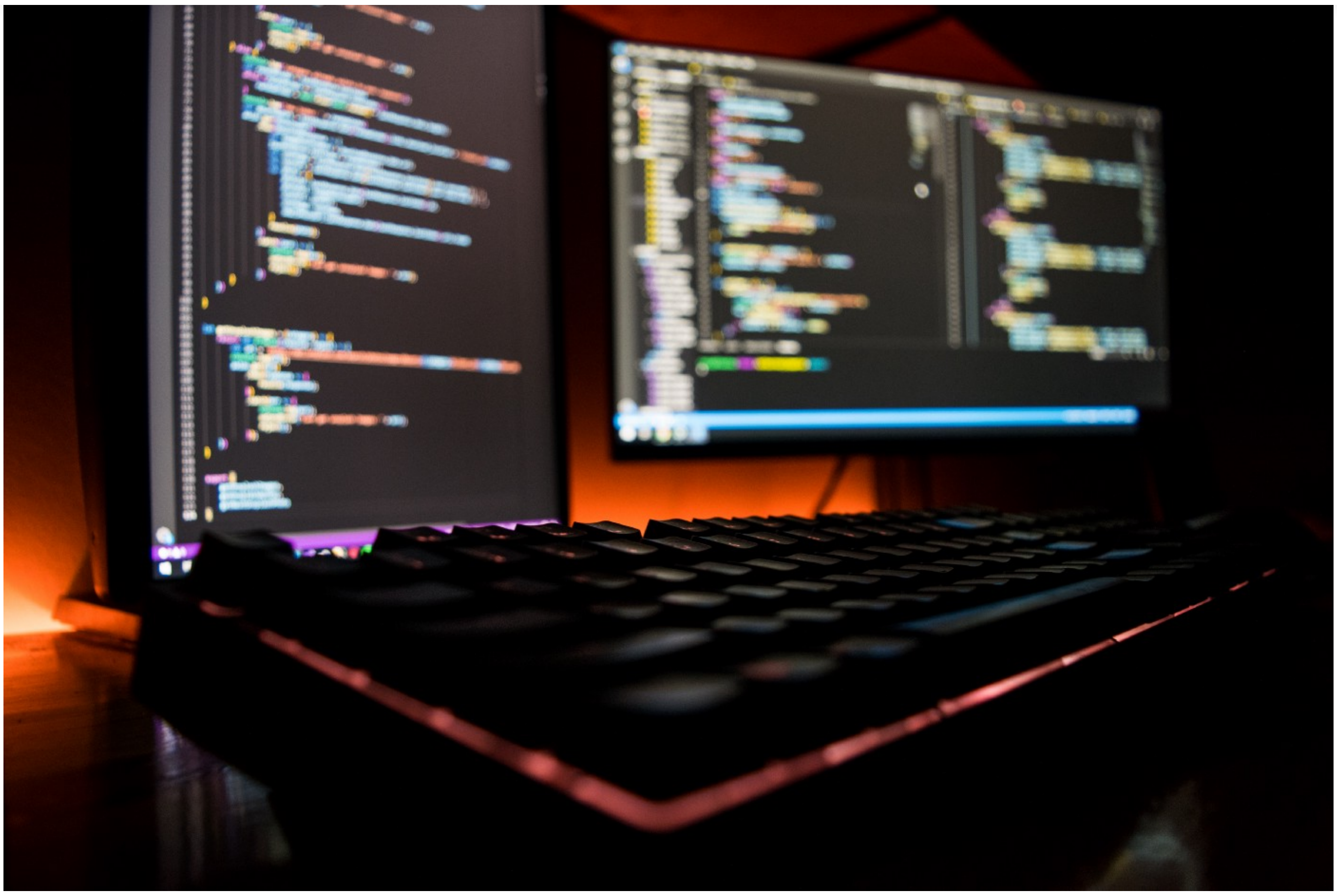


Photo by Fotis Fotopoulos on Unsplash

With this in mind, I plan to compare the pros and cons of using Golang to build a simple ML pipeline. I will simultaneously use Python as a reference point. Also, I will provide my personal opinion on the language in general and to evaluate whether Go has a future within the AI/ML community. So, let's dive right into it.

Note: Before we proceed, this post will not cover how to install and setup Go in your machine. If you have not already done so, please follow these comprehensive instructions.

Table of Content:

Setup

First, we will need to install the following packages in our terminal.

1. Install **DataFrame** and **GoNum** packages and their dependencies. DataFrame and GoNum are similar to Python's NumPy and are often used to manipulate DataFrame objects.

```
$ go get -u github.com/kniren/gota/dataframe
$ go get -u github.com/gonum/matrix
$ go get -u gonum.org/v1/gonum/...
```

2. Install **GoLearn** package. GoLearn is a machine learning library in GoLang that is similar to Python's sklearn. It allows for easy matrix manipulation, construction of ML algorithms, model fitting, and even data splitting for training/evaluation processes.

```
$ g++ --version # make sure you have a G++ compiler

$ go get github.com/gonum/blas

$ go get -t -u -v github.com/sjwhitworth/golearn
$ cd $GOPATH/src/github.com/sjwhitworth/golearn
$ go get -t -u -v ./...
```

Now that the hard part is behind us, let us proceed to the more fun parts!

DataFrame

We will be using the IRIS dataset that describes the different types of iris flower. You can get the IRIS dataset by running the following in your terminal:

Ensure that you place the .CSV file inside the folder you are currently working on and where your *main.go* file is located.

We would need to first import the following packages into our *main.go* program file.

```go
package main

import (
	"fmt"
	"log"
	"os"

	"github.com/kniren/gota/dataframe"

	"github.com/sjwhitworth/golearn/base"
	"github.com/sjwhitworth/golearn/evaluation"
	"github.com/sjwhitworth/golearn/knn"
)
```

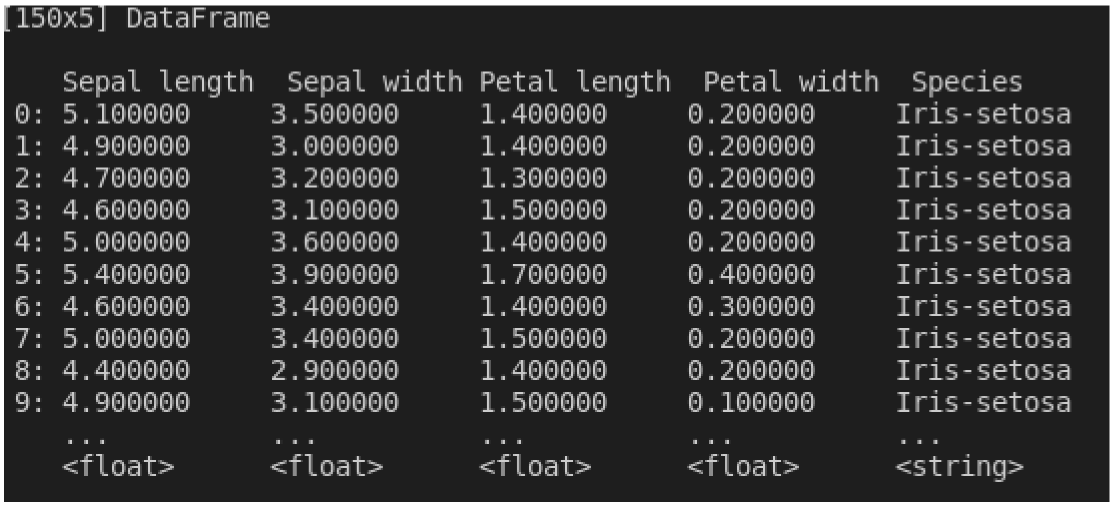
go_ml1.go hosted with ❤ by GitHub view raw

Now that we have all the imports, let's write the main() function to load the CSV data and print them out nicely.

```go
package main

import (
	"fmt"
	"log"
	"os"

	"github.com/kniren/gota/dataframe"

	"github.com/sjwhitworth/golearn/base"
	"github.com/sjwhitworth/golearn/evaluation"
	"github.com/sjwhitworth/golearn/knn"
)

func main() {
	irisCsv, err := os.Open("./iris_headers.csv")
	if err != nil {
```

```
23  }
```

You will have the following nice summary of your data describes the number of rows (150), the number of attributes (5), the attribute summary (including the data type), and the summary view for the first few rows.



```
[150x5] DataFrame

    Sepal length  Sepal width Petal length  Petal width  Species
 0: 5.100000     3.500000     1.400000     0.200000     Iris-setosa
 1: 4.900000     3.000000     1.400000     0.200000     Iris-setosa
 2: 4.700000     3.200000     1.300000     0.200000     Iris-setosa
 3: 4.600000     3.100000     1.500000     0.200000     Iris-setosa
 4: 5.000000     3.600000     1.400000     0.200000     Iris-setosa
 5: 5.400000     3.900000     1.700000     0.400000     Iris-setosa
 6: 4.600000     3.400000     1.400000     0.300000     Iris-setosa
 7: 5.000000     3.400000     1.500000     0.200000     Iris-setosa
 8: 4.400000     2.900000     1.400000     0.200000     Iris-setosa
 9: 4.900000     3.100000     1.500000     0.100000     Iris-setosa
    ...          ...          ...          ...          ...
    <float>      <float>      <float>      <float>      <string>
```

IRIS Dataframe Summary (Image From Author)

Pretty neat! Most of these operations so far are similar to how you would code in Python.

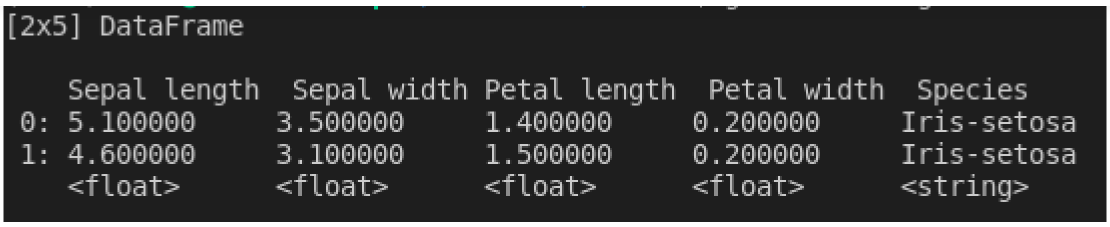
Data Manipulation

Now let's explore how to perform data manipulation in Go.

1. Subsetting

One of the easiest subsetting operations in Python is using the **df.head()** operation. We can similarly perform this function in Go.

```
head := df.Subset([]int{0, 3})
fmt.Println(head)
```



```
[2x5] DataFrame

   Sepal length  Sepal width Petal length  Petal width  Species
0: 5.100000      3.500000    1.400000      0.200000     Iris-setosa
1: 4.600000      3.100000    1.500000      0.200000     Iris-setosa
   <float>       <float>     <float>       <float>      <string>
```

Subsetting a Dataframe in Go (Image From Author)

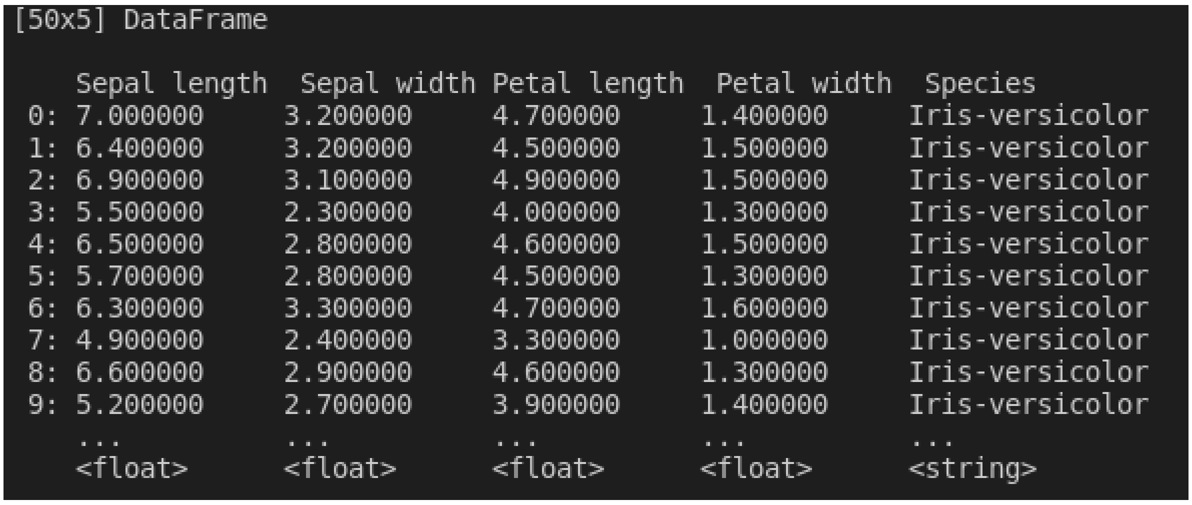
Personally, the operation is not as verbose as that in Python. Without a prior understanding of datatypes in Go, you would have a hard time making sense of this function.

2. Filtering

Suppose now you want to explore the attributes for Iris-versicolor species only. You can filter the rows using the function Filter() as follows.

```
versicolorOnly := df.Filter(dataframe.F{
    Colname:    " Species",
    Comparator: "==",
    Comparando: "Iris-versicolor"
})

fmt.Println(versicolorOnly)
```

You will retrieve rows with Iris-versicolor species only!



```
[50x5] DataFrame

   Sepal length  Sepal width Petal length  Petal width  Species
0: 7.000000      3.200000    4.700000      1.400000     Iris-versicolor
1: 6.400000      3.200000    4.500000      1.500000     Iris-versicolor
2: 6.900000      3.100000    4.900000      1.500000     Iris-versicolor
3: 5.500000      2.300000    4.000000      1.300000     Iris-versicolor
4: 6.500000      2.800000    4.600000      1.500000     Iris-versicolor
5: 5.700000      2.800000    4.500000      1.300000     Iris-versicolor
6: 6.300000      3.300000    4.700000      1.600000     Iris-versicolor
7: 4.900000      2.400000    3.300000      1.000000     Iris-versicolor
8: 6.600000      2.900000    4.600000      1.300000     Iris-versicolor
9: 5.200000      2.700000    3.900000      1.400000     Iris-versicolor
   ...           ...         ...           ...          ...
   <float>       <float>     <float>       <float>      <string>
```

Filtering by Species name (Image From Author)

I, personally, do not mind the change in syntax because these operations so far seem to be pretty intuitive. But the number of words for these programs is significantly longer than that in Python as illustrated below.

```
versicolorOnly = df[df[" Species"] == "Iris-versicolor"]
```

3. Column Selection

In addition to subsetting and rows filtering, you can also subset columns by following this simple operation.

```
attrFiltered := df.Select([]string{"Petal length", "Sepal length"})
fmt.Println(attrFiltered)
```

There are a bunch of other data wrangling operations that you can find, including joins, aggregations, function applications (recall .apply() in pandas?), *etc.*

Overall, I would still prefer Python, but maybe because I do not have sufficient practice with Go. The verbosity of Go, however, might dissuade me from pursuing the former.

Machine Learning in Go

Now that we have some handle on manipulating Dataframe in Go, we can start building a simple Machine Learning pipeline. For this example, let us build a simple KNN classifier to determine IRIS' species type given its collection of attributes (eg. Petal length, petal width, etc).

0. Reload the Data

In the main() function, let's reload our CSV dataset.

```
fmt.Println("Load our csv data")
```

1. Initialize a KNN Classifier

Here, we initialize a new KnnClassifier from the *knn subpackage of GoLearn*. We then pass in the necessary attributes to the new classifier class: using *euclidean* as the distance function, *linear* as its algorithmic kernel, and *2* as the number of neighbors of choice.

```
fmt.Println("Initialize our KNN classifier")
cls := knn.NewKnnClassifier("euclidean", "linear", 2)
```

2. Training-Testing Split

As usual, we perform a train/test data split (50% each). Nothing fancy so far, everything seems identical to how we would do it in Python's sklearn library.

```
fmt.Println("Perform a training-test split")
trainData, testData := base.InstancesTrainTestSplit(rawData, 0.50)
```

3. Train the Classifier

The main part of any ML pipeline would be the training of the classifier. But, if you are familiar with Python's sklearn package, then the process in Go is similar if not identical.

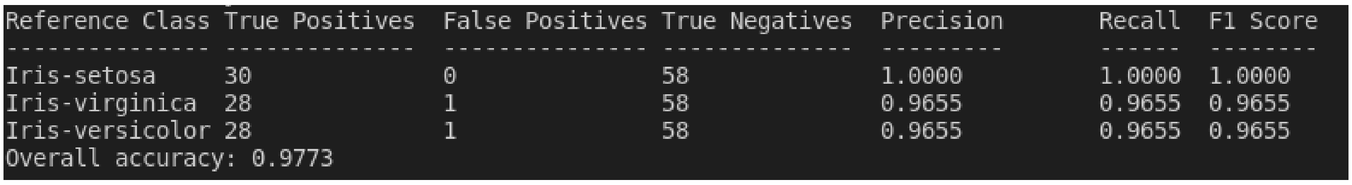
```
cls.Fit(trainData)
```

4. Summary Metrics

After the training step has been done, we can perform an evaluation using the kept-out test data and retrieve the performance summary.

```
fmt.Println("Print our summary metrics")

confusionMat, err := evaluation.GetConfusionMatrix(testData,
fmt.Println(evaluation.GetSummary(confusionMat))
```

If everything is going smoothly, you would have the following printed out in your terminal!



```
Reference Class True Positives  False Positives True Negatives  Precision       Recall  F1 Score
--------------- --------------  --------------- --------------  ---------       ------  --------
Iris-setosa     30              0               58              1.0000          1.0000  1.0000
Iris-virginica  28              1               58              0.9655          0.9655  0.9655
Iris-versicolor 28              1               58              0.9655          0.9655  0.9655
Overall accuracy: 0.9773
```

Evaluation Summary of our KNN classifier (Image From Author)

Not too bad! We achieve a 97.73% accuracy for our newly trained KNN classifier. Try and implement other types of classifiers you can find in GoLearn!

Finally, you can find the full code in the snippet below!

Conclusion

Overall, I find the Go syntax for building a Machine Learning pipeline to be sufficiently intuitive. However, I find that the size of Go's ML community is smaller than that in Python. This makes troubleshooting cumbersome, if not frustrating at times. In addition, the lack of GPU support in many of these packages can impede your AI development process. Not only that but there seems to be very little interoperability between the different ML-related packages. For instance, the GoLearn package implements its own Dataframe "Instance" class that may not work well with the native GoNum's class or even GoTA's DataFrame object.

Nonetheless, Go, with its superb speed and concurrency, has the potential to overtake Python in Machine Learning applications; but it needs to have a sufficient number (ie. critical mass) of AI developers to do so.

Do subscribe to my Email newsletter: *https://tinyurl.com/2npw2fnz* ***where I regularly summarize AI research papers in plain English and beautiful visualization.***

References

[1] https://en.wikipedia.org/wiki/Go_(programming_language)

[2] GoNum: https://github.com/gonum/gonum

[3] GoLearn: https://pkg.go.dev/github.com/Soypete/golearn

[4] GoTA: https://github.com/go-gota/gota