WHITEPAPER

Pipelining Machine Learning Models Together

ALGORITHMIA

Table of Contents

About Algorithmia

Algorithmia automates, optimizes, and accelerates every step of the journey to deploying of AI/ML at scale. We allow anyone to run models on massively parallel infrastructure in minutes instead of months. In our cloud or your datacenter - all completely managed for maximum performance at minimum cost. Already trusted by over 60k developers and major enterprise customers, Algorithmia makes scalable Machine Learning fast, simple, and cost-effective for everyone.

Democratizing access to algorithmic intelligence.

Sign-Up for Free or Schedule a Demo Today

Pipelining Your Machine Learning Models Together

This whitepaper walks through Machine Learning pipelining, an efficient way of splitting up your ML models into independent parts. We'll explain what pipelining is, why it matters, some sample use cases, and how Algorithmia makes it easy to pipeline your algorithms together.

The Definition of Pipelining

Pipelining is the art of splitting up your Machine Learning workflows into independent, reusable, modular parts.

Pipelining in Machine Learning is just like the recent push for Microservices based architectures. The broad idea is that by splitting your application into basic and siloed parts, you can build more powerful software over time. The Linux and Unix operating systems are also founded on this principle: functions like `grep` and `cat` are all basic in of themselves, but by piping them together you can create impressive functions.

Why Pipelining is So Powerful

To help understand why pipelining can make such a big difference in performance and design, consider a typical Machine Learning workflow.

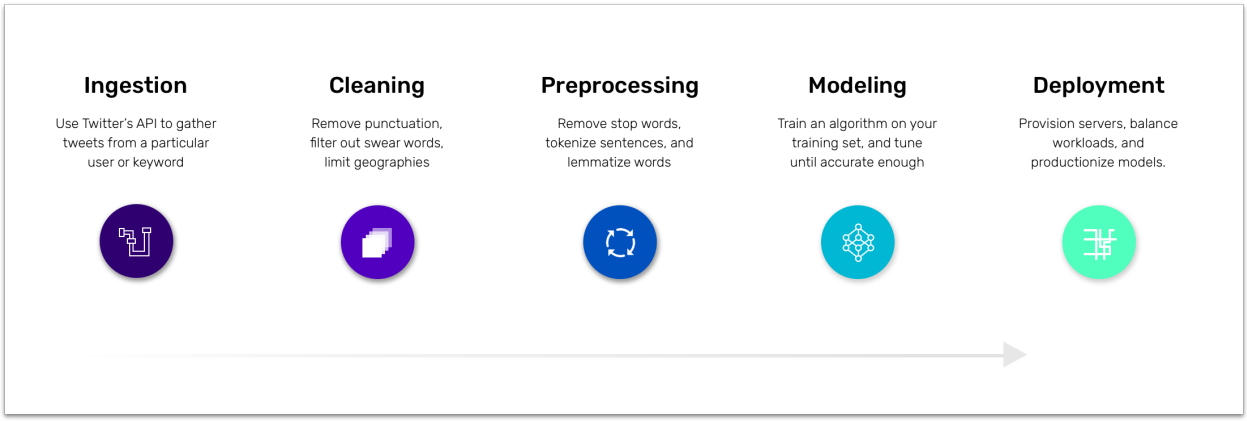

Figure 1: Typical Machine Learning Workflow

In a run-of-the-mill system design, a Data Scientist or Engineer might run all of these tasks together in a monolith, like a Jupyter Notebook as a sandbox or as a script in production. The same script will extract data, clean and prepare it, model it, and get it deployed. Machine Learning models will generally contain far less code than your average software application, so this approach makes sense: it keeps all of your assets in one spot and makes it easy to see how your flows work together.

Once you introduce scale into the equation though, monolithic Machine Learning workflows tend to fall apart.

Three significant problems arise with this kind of setup as you grow:

- **Volume:** if your deployment involves multiple versions of the same model, you'll be running a workflow twice even though the results of ingestion and preparation aren't changing.
- **Variety:** as you expand your model portfolio, you'll be copying and pasting code from the early stages of your workflow. That's inefficient, and a no-no in software development.
- **Versioning:** any time you change the configuration of your data sources or any other part of the workflow, you'll need to manually update all of your scripts. That can become infeasible and destructive as your business grows.

With pipelining, each part of your workflow is abstracted into a service – often with an API endpoint – that operates independently. For each workflow you design, you can pick and choose which elements you need and utilize them: but the master service resides in one place.

This kind of architecture alleviates all of the difficulties that a monolithic design presents:

- **Volume:** only call parts of the pipeline when you need them, and cache or store results that you plan on reusing (that can be its own service, too).
- **Variety:** as your model portfolio expands, you can still use the same tools from the earlier parts of your pipelines without useless replication.
- **Versioning:** when deployed as elements, there's only one service for you to change and version properly. Any workflows that call the service will automatically incorporate your updates.

Performance and Organizational Benefits of Pipelining

Scheduling and Runtime Optimization

Pipelining can also help your Machine Learning projects in the performance arena. As your teams grow, you'll find that some parts of your pipeline get heavily reused: and if you know that in advance, you can tweak your deployments to optimize for algorithm-to-algorithm calls. Concretely, if you know in advance which algorithms will call which other algorithms, you can get the right ones running to reduce your compute time and avoid cold-starts.

Language and Framework Agnosticism

Another benefit of pipelining is the power to be language and framework agnostic. When your Machine Learning workflow is monolithic, you'll need to be consistent in the programming language you use and load all of your dependencies together. But as a pipeline with API endpoints, your services can be written in independent languages and utilize their own frameworks. Since Machine Learning workflows will typically span across multiple parts of a given technology stack, the ability to use different languages with different strengths is key.

Broader Applicability and Fit

Pipelining can also be viewed as a connector. If your organization has multiple algorithms - all with their own input requirements – you need to be specific about what you pass to them. But if you have independently operating services, you can transform your inputs on the fly to fit the algorithms you want to use. An example is splitting up a video into images so you can use image classification algorithms, a process we'll explore later.

Some practical use cases will drive home the point of why this pipelining paradigm is so important for scaling teams.

Practical Use Case: Twitter Sentiment Analysis

Tasks in Natural Language Processing often involve multiple, repeatable steps. Take a look at this example of an NLP workflow that downloads tweets to analyze their sentiment.

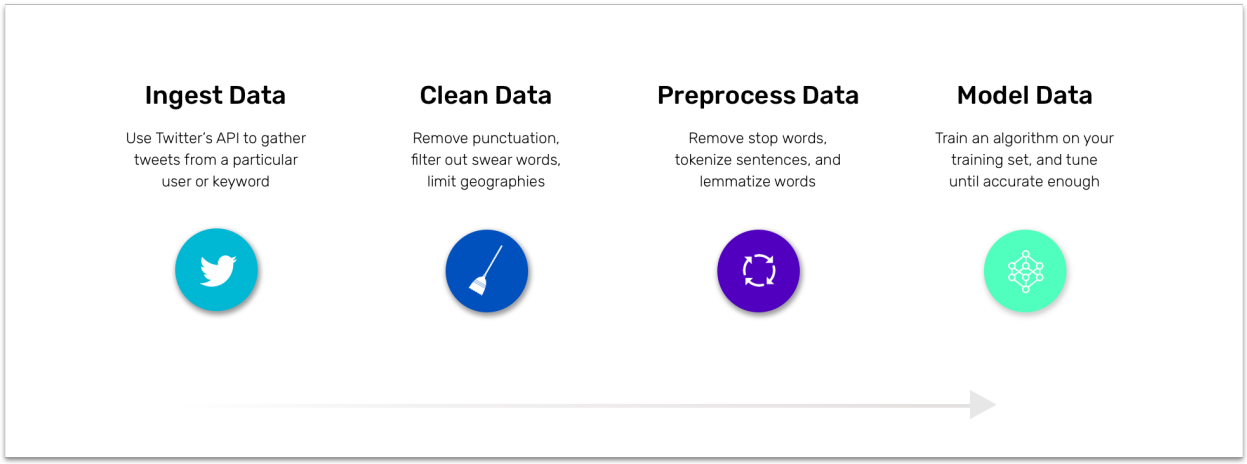

Figure 2A: Twitter Sentiment Analysis Workflow

Data is ingested from Twitter using their API, cleaned for punctuation and whitespace, tokenized and lemmatized, and then passed through a sentiment analysis algorithm that classifies the tweet.

Keeping all of these functions together might make sense at first, but as you begin to apply more analysis to your base data (the tweets, in this case) it makes sense to modularize your workflow. Here's what that same system might look like with independent, pipelined components:

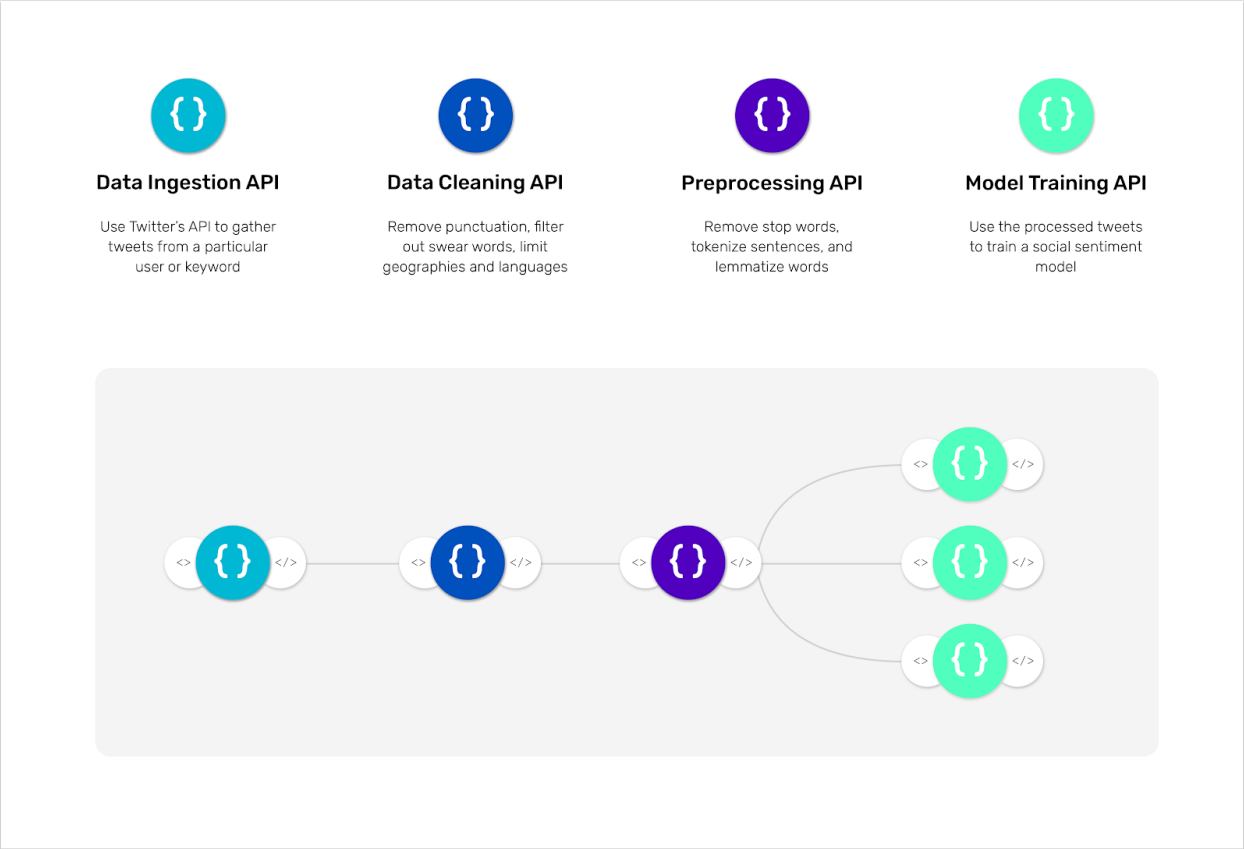

Figure 2B: Twitter Sentiment Analysis Pipeline

With this architecture, it's easy to swap out the algorithm you're using for another, change the lemmatizer, or grab tweets from a different user – all without breaking or changing the other elements of your workflow.

Practical Use Case: Video Transformation

Videos are just a rapid series of images, and so the algorithms that classify and transform images can do the same for video. A typical workflow involves splitting a video into images, running image transformation algorithms on each one, and then re-combining them into a now transformed video.

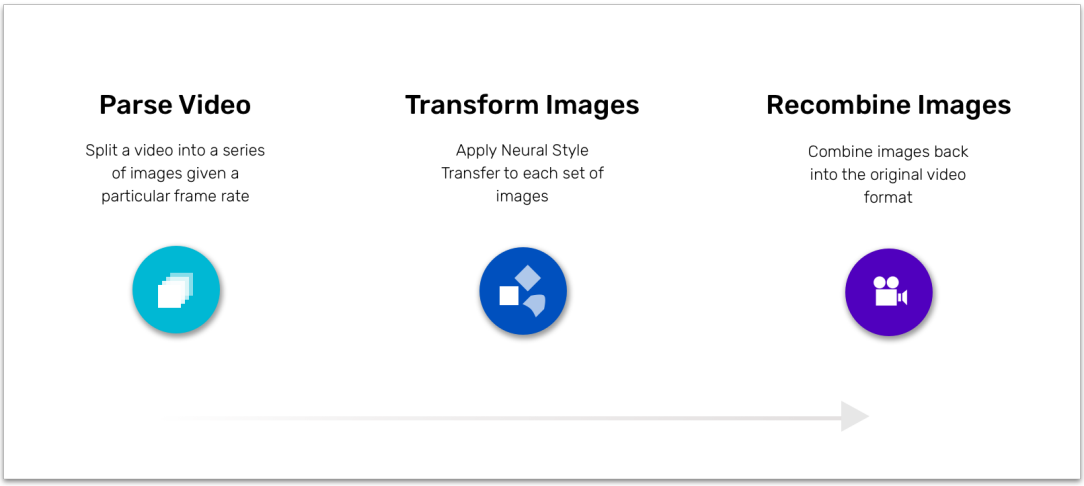

Figure 3A: Video Transformation Workflow

The video is split into the specified number of frames per second, run through neural style transfer, and then re-combined to form a fully transformed video.

For a video (or videos) on the shorter side, this is a workable flow. But as videos get larger and models get more complex, parallel processing becomes more and more important. Pipelining this workflow might separate these 3 elements into individual services, and have them call each other.

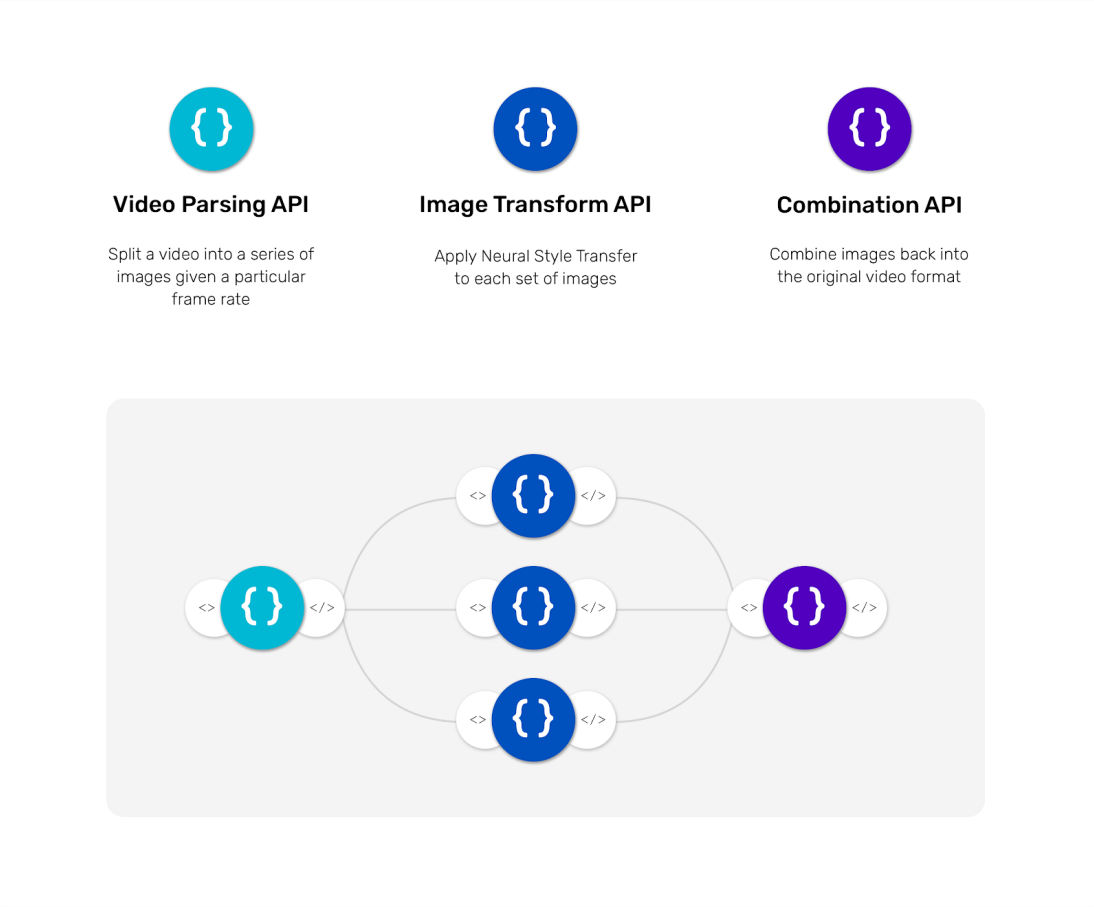

Figure 3B: Video Transformation Pipeline

With this process organized modularly, you can recursively call the splitting algorithm on videos of arbitrary length and transform them easily. You'll also have the freedom to process your images in parallel by sending them to multiple endpoints, which is much more difficult in a monolithic script.

Pipelining on Algorithmia

Algorithmia's AI Layer was built from the ground up with these kinds of use cases in mind. Pipelining is a key part of any full scale deployment solution: Data Science teams need to be able to productionize their models as parts of a whole.

With the AI Layer, pipelining is simple:

- Algorithms are packaged as **microservices with API endpoints**: calling any algorithm or function is as easy as `algorithm.pipe(input)`
- The AI Layer supports most **major languages and frameworks**, so your pipeline can be input agnostic
- When uploading a model, you can **set permissions** and choose to allow it to call other algorithms

```
#Initialize client and algorithms
client = Algorithmia.client('YOUR_API_KEY')
algo1 = client.algo('username/cleaning')
algo2 = client.algo('username/preprocessing')


#Pipe algorithm inputs and outputs
print(algo2.pipe(algo1.pipe(input)).result)
```

A lot of the important work in pipelining is happening on the backend, too. When you define your pipeline, the AI Layer is optimizing scheduling behind the scenes to make your runtime faster and more efficient. If one algorithm of yours consistently calls another, the system will pre-start your dependent models to reduce compute time and save you money.

See It For Yourself

Pipelining is just one part of the capabilities that the AI Layer brings to the table. Set up a demo with one of our Solutions Architects to see how you can automate your DevOps for Machine Learning.

Sign-Up for Free or Schedule a Demo Today