An approach to Parameterized, Distributed, Heterogeneous Machine Learning

Ashwin Nimhan Student of Data Science
Indiana University Bloomington
animhan@indiana.edu

Manashree Rao Student of Data Science
Indiana University Bloomington
manarao@indiana.edu

Abstract

Keywords: Data Pipeline, Auto ML, Black-box ML, predictive modelling

I. INTRODUCTION

Machine learning has achieved considerable successes in recent years and an ever-growing number of disciplines rely on it. However, this success crucially relies on human machine learning experts, who select appropriate features, workflows, machine learning paradigms, algorithms, and their hyperparameters. The research area that targets progressive automation of machine learning is AutoML. The goal is to design the perfect machine learning black box capable of performing all model selection and hyper-parameter tuning without any human intervention. The current approaches in the AutoML field are heavily dependent on underlying platform and supported languages, like auto-sklearn or auto-weka. What we want to achieve is try to scale this across multiple programming languages, i.e., use python, R, Spark ML, etc. together for ML tasks like choosing a machine learning model, tuning hyper-parameters, avoiding overfitting and optimization for a provided evaluation metric.

II. SURVEY OF EXISTING SYSTEMS

A. AutoCompete

AutoCompete proposes a system that automates a lot of the classical machine learning cycle and tries to build a predictive model without (or with a very little) human interference. AutoCompete works only with datasets in tabular format. The most important components of the proposed AutoCompete system are the ML Model Selector and Hyper-parameter Selector. In addition to these, there is a data splitter, data type identifier, feature stacker, decomposition tools and feature selector. If a classification task is encountered, the dataset is split in a stratified manner, such that both the training and validation set have the same distribution of labels. The validation set is always kept separate from any transformations being used on the training set and is not touched at any point in the pipeline. All the transformations on the training set are saved and then applied on the validation set in the end. This ensures that the system is not over-fitting and the models thus produced as a result of the AutoCompete pipeline generalize on unseen datasets. Once the splitting is done, the type of features are identified and appropriate transformations are used. Each transformation is then fed through a feature selection mechanism which in turn sends the selected features and the transformation pipeline through model selector and hyper-parameter selector. The transformation and the model with the best performance is used in the end.

B. Auto-Sklearn

Auto-sklearn provides out-of-the-box supervised machine learning. Built around the scikit-learn machine learning library, auto-sklearn automatically searches for the right learning algorithm for a new machine learning dataset and optimizes its hyperparameters. Thus, it frees the machine learning practitioner from these tedious tasks and allows her to focus on the real problem. Auto-sklearn extends the idea of configuring a general machine learning framework with efficient global optimization which was introduced with Auto-WEKA. To improve generalization, auto-sklearn builds an ensemble of all models tested during the global optimization process. In order to speed up the optimization process, auto-sklearn uses meta-learning to identify similar datasets and use knowledge gathered in the past. Auto-sklearn wraps a total of 15 classification algorithms, 14 feature preprocessing algorithms and takes care about data scaling, encoding of categorical parameters and missing values.

C. Auto-WEKA

Auto-WEKA, a system designed to help such users by automatically searching through the joint space of WEKAs learning algorithms and their respective hyperparameter settings to maximize performance, using a state-of-the-art Bayesian optimization method. Many different machine learning algorithms exist that can easily be used off the shelf in the open source WEKA package. However, each of these algorithms have their own hyperparameters that can drastically change their performance, and there are a staggeringly large number of possible alternatives overall. Auto-WEKA considers the problem of simultaneously selecting a learning algorithm and setting its hyperparameters, going beyond previous methods that address these issues in isolation. Auto-WEKA does this using a fully automated approach, leveraging recent innovations in Bayesian optimization. Auto-WEKA helps non-expert users to more effectively identify machine learning algorithms and hyperparameter settings appropriate to their applications, and hence to achieve improved performance.

D. *Hyperopt-Sklearn*

Hyperopt-Sklearn: is a project that brings the benefits of automatic algorithm configuration to users of Python and scikit-learn. Hyperopt-sklearn is Hyperopt-based model selection among machine learning algorithms in scikit-learn. Hyperopt-Sklearn uses Hyperopt to describe a search space over possible configurations of Scikit-Learn components, including preprocessing and classification modules. The Hyperopt library offers optimization algorithms for search spaces that arise in algorithm configuration. These spaces are characterized by a variety of types of variables (continuous, ordinal, categorical), different sensitivity profiles (e.g. uniform vs. log scaling), and conditional structure (when there is a choice between two classifiers, the parameters of one classifier are irrelevant when the other classifier is chosen).

III. LIMITATIONS OF EXISTING SYSTEMS

Above mentioned frameworks automate in bits and pieces steps in the generic machine learning pipeline. These popular machine learning face the limitations of language of implementation and availability of optimized package in the ecosystem. Optimized libraries written in C, C++, CUDA and other low level languages, generally utilized in production which may or may not run on commodity harware, require special configurations and may lack interface for popular languages like python.

As a result of these limitations it becomes almost impossible to build a hetergenous pipeline by choosing high performance componenets from different frameworks and leveraging specilized configured and high-performant hardware like HPC Systems.

To resolve this, we want to utilize a Workflow automation framework to merge and stitch together heterogeneous hardware and ML libraries so that task can be run in a distributed manner on specialized remote hardware and then can be synchronized using DAG based systems. We intend to leverage the existing work put in this libraries and new advances in GPU based learning along with frameworks like Spark and Hadoop which are common in production while still maintianing flexibility to accomodate new and upcoming libraries without the limitation of the hardware, language or availability of programming interfaces in popular languages or robust APIs.

IV. USING WORKFLOW MANAGEMENT SYSTEMS FOR ML PIPELINES

A. *General requirements*

We list the requirements we think would be useful for desiging complex ML systems while supporting failover and task rescheduling for individual tasks which are commonly seen patterns in production below.

- Central Repository
- Support for dynamic DAG mechanism to handle complex task dependencies, synchronization between parallel tasks and notifications and interact with systems that operate on Data - Hive/Presto/HDFS/Postgres/S3 etc
- Keep track of the Operations and the metrics of the workflow, monitor the current/historic state of the jobs, the results of the jobs etc. Ensure Fault tolerance of the pipelines and have the capability to back fill any missing data, etc. and transfer and storage of intermediate results.
- Parameterization of each step in the machine learning pipeline.
- Support for multiple programming environments (python, R, CUDA, etc) and libraries (sklearn, MLLib, Mahout, tensorflow, keras, caffe, Vowpal-Wabbit, etc).

B. *Survey of popular Workflow Management Systems*

There are different workflow management systems like:

- Luigi - Luigi is a Python module that helps you build complex pipelines of batch jobs.
- Airflow - Airflow is a platform to programmatically author, schedule and monitor workflows.
- Pinball - Pinball is a scalable workflow manager developed at Pinterest.
- Azkaban - Azkaban is a batch workflow job scheduler created at LinkedIn to run Hadoop jobs.
- Oozie - Oozie is a workflow scheduler system to manage Apache Hadoop jobs

We compare these systems as per our requirements.

Feature	Luigi	Airflow	Pinball
Data Pipeline	Tasks are grouped together into a DAG to be run. Most of the code treats Tasks as the main unit of work.	DAG (Directed Acyclic Graph) is used to define Jobs.	Workflow
Class processing the main unit of work	Tasks/Workers	Operators	Jobs/Workers
UI	Overview of Tasks only	Comprehensive, with multiple screens	Detailed, looks like Sidekiq
metadata/job status	Task status is stored in database. Similar to Airflow, but fewer details.	Job status is stored in a database. Operators mark jobs as passed or failed. Last updated is refreshed frequently with a heartbeat function. kill_zombies() is called to clear all jobs with older heartbeats.	Workers 'claim' messages from the queue with an ownership timestamp on the message. This lease claim gets renewed frequently. Messages with older lease claims are requeued. Messages successfully processed are archived to S3 file system using Secor. Job status is stored to database.
scaling	Create multiple Tasks	DAGs can be constructed with multiple Operators. Scale out by adding Celery workers	Add Workers
parallel execution	Subprocess	Subprocess	Threading
dependency management	Tasks can be constructed with requires() method	Operators can be constructed with depends_on_past parameter	Jobs can require other jobs to finish first before starting, eg child_job requires parent_job.
Code	Code	Code	Python dict+code
state persistence	uses SQLAlchemy for abstracting away the choice of and querying the database(Mysql or Postgresql)	It supports any store that is supported by SQL Alchemy. If you don't use a external store, the state is not saved (only log file). The status of a task is always checked based on the existence of its output.	Yes to db
tracks history	Yes to db	Yes	Yes to db
messaging queue/message broker	No	Celery and RabbitMQ/Reddis	No
fault tolerance	No	Yes	Yes
hadoop	yes	yes	yes
pig	yes	yes	no
hive	yes	yes	yes
pgsql	yes	yes	no
mysql	yes	yes	no
redshift	no	yes	no
s3	yes	yes	yes

V. AIRFLOW

Airflow is a platform to programmaticaly author, schedule and monitor data pipelines. A few basic components of Airflow are:

- The job definitions, in source control.
- A rich CLI (command line interface) to test, run, backfill, describe and clear parts of your DAGs.
- A web application, to explore your DAGs definition, their dependencies, progress, metadata and logs. The web server is packaged with Airflow and is built on top of the Flask Python web framework.
- A metadata repository, typically a MySQL or Postgres database that Airflow uses to keep track of task job statuses and other persistent information.

- An array of workers, running the jobs task instances in a distributed fashion.
- Scheduler processes, that fire up the task instances that are ready to run.

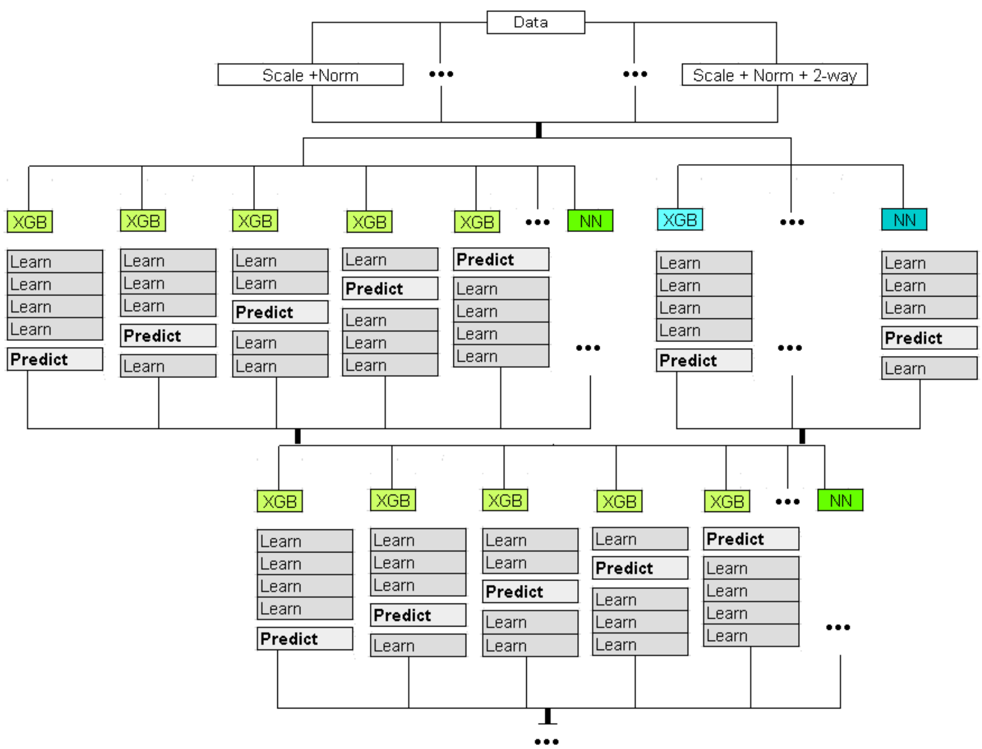
Why Airflow

Apache Airflow supports the folowing operations making it the ideal choice of the workflow systems surveyed above:

- Airflow pipelines are configuration as code (Python), allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically.
- Define your own operators, executors and extend the library so that it fits the level of abstraction that suits your environment.
- Parameterizing your scripts is built into the core of Airflow using the powerful Jinja templating engine.
- Airflow has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. Airflow is ready to scale to infinity and to hetergenous systems and specially configured environments through workers and operators.
- Components like RabbitMQ, MySQL/Postgres and Celery commponly deployed by big internet companies in production can be easily leveraged.

VI. Architecture of our System

Fig. 1: Generic Architecture



Setup tasks - We need the following steps to be completed before executing the DAGs:

- All nodes have Airflow installed on it and are configured as per topology.
- All scripts share a storage area where intermediate files are written.
- Programming environment(python, R) is setup alongwith required libraries(scikit-learn, keras, xgboost, etc.)

Once setup is done, we proceed as follows:-

- Data is downloaded from the given repository link and placed in the shared storage.
- Each layer is executed sequentially and the layers below L_i depend on the results from the layers above L_{i-1}. Results from each layer are written to the shared location.
- Each fork in the layer can be executed independently. Each fork can be modelled as an independent airflow task and generic script and respective parameters and data can be fetched by the worker from the centralized shared storage space. Airflow supports dynamic tasks and parameter using jinja templating.
- Multiple layers can be stacked as per requirement using Airflow which supports complex DAGs.

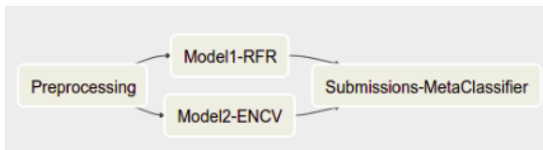
VII. Usecase - Allstate Claims prediction challenge

We try to model the kaggle dataset provived by Allstate, one of the largest insurance companies in the United State which offers insurance on vehicles, home, property, condo, renters insurance, etc. In order to provide better claims service for Allstates customers, the company is developing automated methods to predict claims serverity. The goal of this challenge is to build a model that can help Allstate to predict the severity of the claims accurately and highlight factors that influence this severity. With this information, Allstate can proposed or adjust more suitable insurance packages for their customers. Each row in this dataset represents an insurance claim.

After data pre-processing and feature engineering in layer 0, the next layer takes this as input for modelling. As a proof of concept we implement the topology on a single node, pseudo distributed airflow installation we implement a simple DAG relying on relatively simple models to keep the computational complexity low.

Currently, we are using a 3 layer model, in which first layer implements data preprocessing the second layer consists of 2 models - Random Forest and Elastic Net Regression model and the third layer is a simple linear combination of the results from layer 1. The results are then verified using the Kaggle challenge evaluation board on unseen test data.

Fig. 2: Airflow DAG generated



The following table summarizes the performance of individual models and the basic ensemble:

Number of Models	Model Names	Score
1	Random Forest	1187.51415
1	Elastic Net Regression	1261.16079
2	Meta-Learner	1175.98815

As we can see, Airflow gives a very fine control on creation of multiple tasks within a DAG while creating reusable building blocks as well as computation frameworks and services. It has accelerated authoring pipelines and reduced the amount of time monitoring and troubleshooting.

References

[1] https://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf
[2] http://bytepawn.com/luigi-airflow-pinball.html
[3] https://www.slideshare.net/r39132/airflow-agari-63072756
[4] https://www.slideshare.net/erikbern/luigi-presentation-nyc-data-science
[5] https://www.slideshare.net/growthintel/a-beginners-guide-to-building-data-pipelines-with-luigi
[6] https://www.michaelcho.me/article/data-pipelines-airflow-vs-pinball-vs-luigi
[7] http://www.ml4aad.org/automl/auto-sklearn/
[8] http://www.kdnuggets.com/2016/08/winning-automl-challenge-auto-sklearn.html
[9] https://airflow.incubator.apache.org

APPENDIX

Code Section 1 - Installation of Airflow

Installing and configuring Apache Airflow

```
Installing and configuring Apache Airflow

Install Dependencies
        apt-get update
        apt-get install unzip
        apt-get install build-essential
        apt-get install python-dev
        apt-get install libsasl2-dev
        apt-get install python-pandas

Install Pip

Install MySQL
        sudo apt-get install mysql-server
        apt-get install libmysqlclient-dev
        pip install MySQL-python

Install RabbitMQ
        apt-get install rabbitmq-server
```

```
Install airflow and required libraries
        pip install airflow=1.7.0
        pip install airflow[mysql]
        pip install airflow[rabbitmq]
        pip install airflow[celery]

Configuring Airflow @ {AIRFLOW_HOME}/airflow.cfg
        executor = CeleryExecutor
        sql_alchemy_conn = mysql://root:root@localhost:3306/airflow
        broker_url = amqp://guest:guest@localhost:5672/
        celery_result_backend = db+mysql://root:root@localhost:3306/airflow

On Master execute following initialization commands
(Initialize the Airflow database, start the web server and scheduler)
        service rabbitmq-server start
        airflow initdb
        airflow webserver
        airflow scheduler
        airflow flower

On Worker execute the following commands
(Initialize Airflow worker)
        airflow worker
```

Code Section 2 - Airflow Task File

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'Team Sayian',
    'depends_on_past': False,
    'start_date': datetime(2017, 5, 5),
    'email': ['animhan@indiana.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5)
}
dag = DAG('allstate', default_args=default_args, schedule_interval=timedelta(1))

# t1, t2 and t3 are examples of tasks created by instantiating operators
t1 = BashOperator(task_id='Preprocessing', bash_command='python allstate-factorize.py', dag=dag)
t2 = BashOperator( task_id="Model1-RFR", bash_command='python allstate-final.py classifier=rfr', retries=3, dag=dag)

t3 = BashOperator( task_id='Model2-ENCV', bash_command='python allstate-final.py classifier=encv', dag=dag)

t4 = BashOperator( task_id='Submissions-MetaClassifier', bash_command='python combine_submission.py files=2 w1=0.9 w2=0.1
f1=rfr_predictions.csv f2=encv_predictions.csv output=submission_1.csv', retries=3, dag=dag)

t2.set_upstream(t1)
t3.set_upstream(t1)
t4.set_upstream(t2)
t4.set_upstream(t3)
```

Code Section 3 - Algorithm Layers

```
args = dict([arg.split('=', maxsplit=1) for arg in sys.argv[1:]])

ESTIMATORS = {
        "encv":    ElasticNetCV(),
        "rfr":     RandomForestRegressor(n_estimators=250),
        "svr":     SVR(C=1.0, epsilon=0.2),
        "gbr":     GradientBoostingRegressor(n_estimators=250),
        "adb":     AdaBoostRegressor(n_estimators=250),
        "knn4":   KNeighborsRegressor(n_neighbors=4)
        # add multiple classifiers/models
}

test_predictions = pd.DataFrame({'id': test_id, 'loss': np.nan})
test_predictions.set_index(['id'])

name = args['classifier']
output = args.get("output",name+'_predictions.csv')

if name in ESTIMATORS.keys():
        estimator = ESTIMATORS[name]
        estimator.fit(train, train_labels)
        test_labels = np.exp(estimator.predict(test))-shift
        test_predictions = test_predictions.assign(loss = test_labels)
        test_predictions.to_csv(output, index=False)
        print("Model: ", name,"output file name: ",output)
```

Code Section 4 - Meta Learner Layer/Combining submissions

```
args = dict([arg.split('=', maxsplit=1) for arg in sys.argv[1:]])

file_count = int(args['files'])
total_weight = 0.0
for i in range(1,file_count+1):
        print('w'+str(i))
        total_weight=total_weight + float(args['w'+str(i)])
if int(total_weight) != 1:
        print("All weights must sum upto 1. Please re-run the file.")
        quit()
else:
        print("All weights sum upto 1")
        for i in range(1,file_count+1):
                if i==1:
                        submission = pd.read_csv(args['f1'])
                        submission['label'] *= float(args['w1'])
                else:
                        submission['label'] += float(args['w'+str(i)]) * pd.read_csv(args['f'+str(i)])['label'].values

        submission.to_csv(args['output'],index=False)
```

Screenshots of Airflow UI (DAGs and task monitoring):

Fig. 3: Homepage for DAG - shows all DAGs with their status

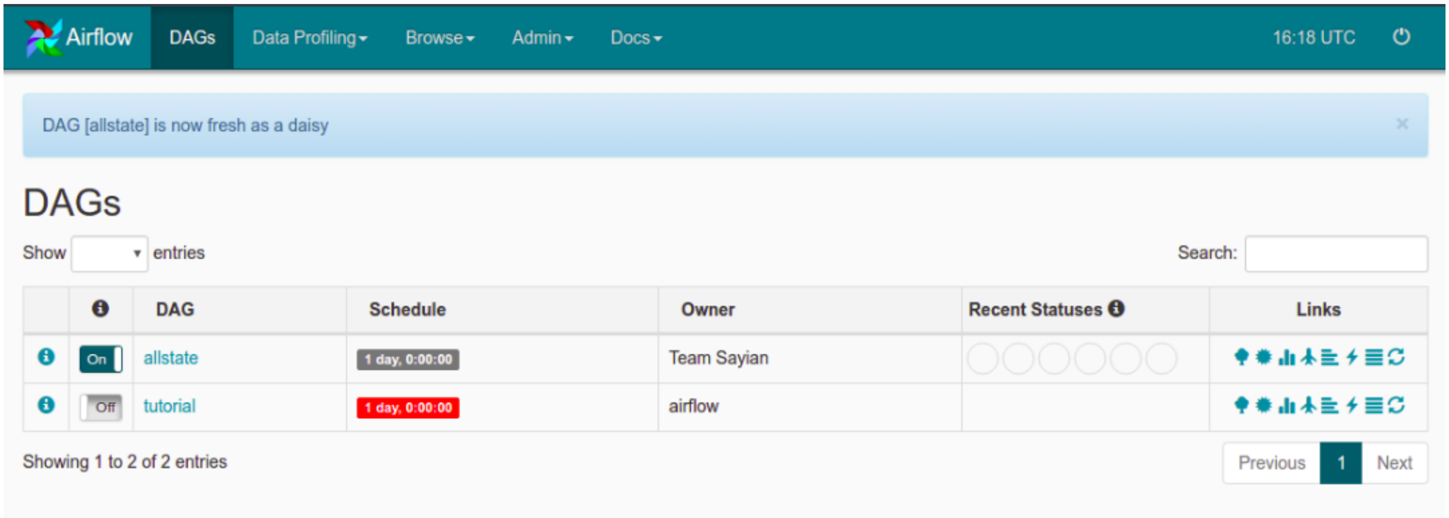


Fig. 4: Status of currently running DAG

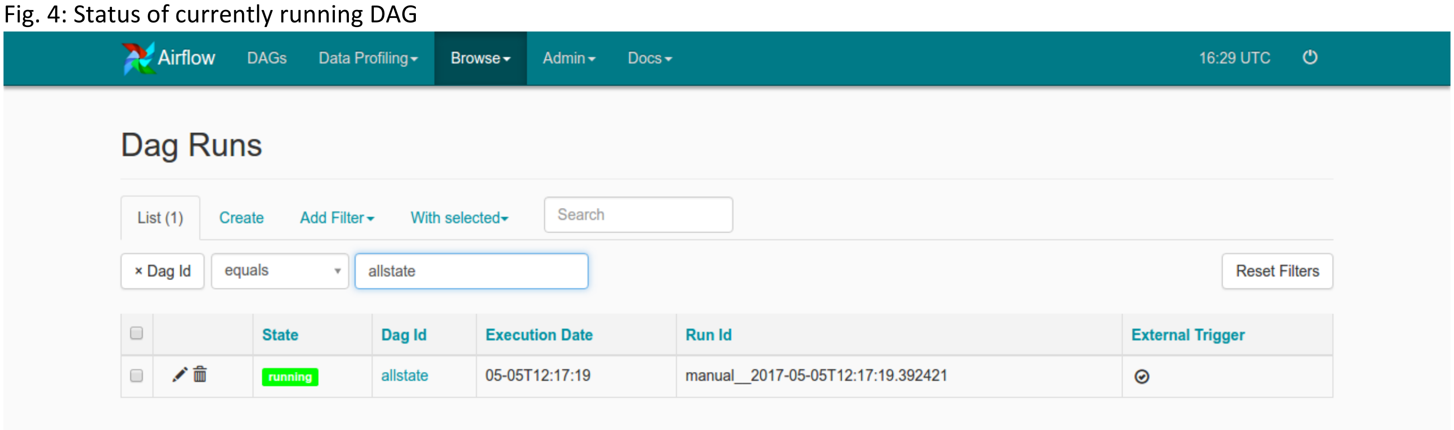


Fig. 5: Details of currently running DAG

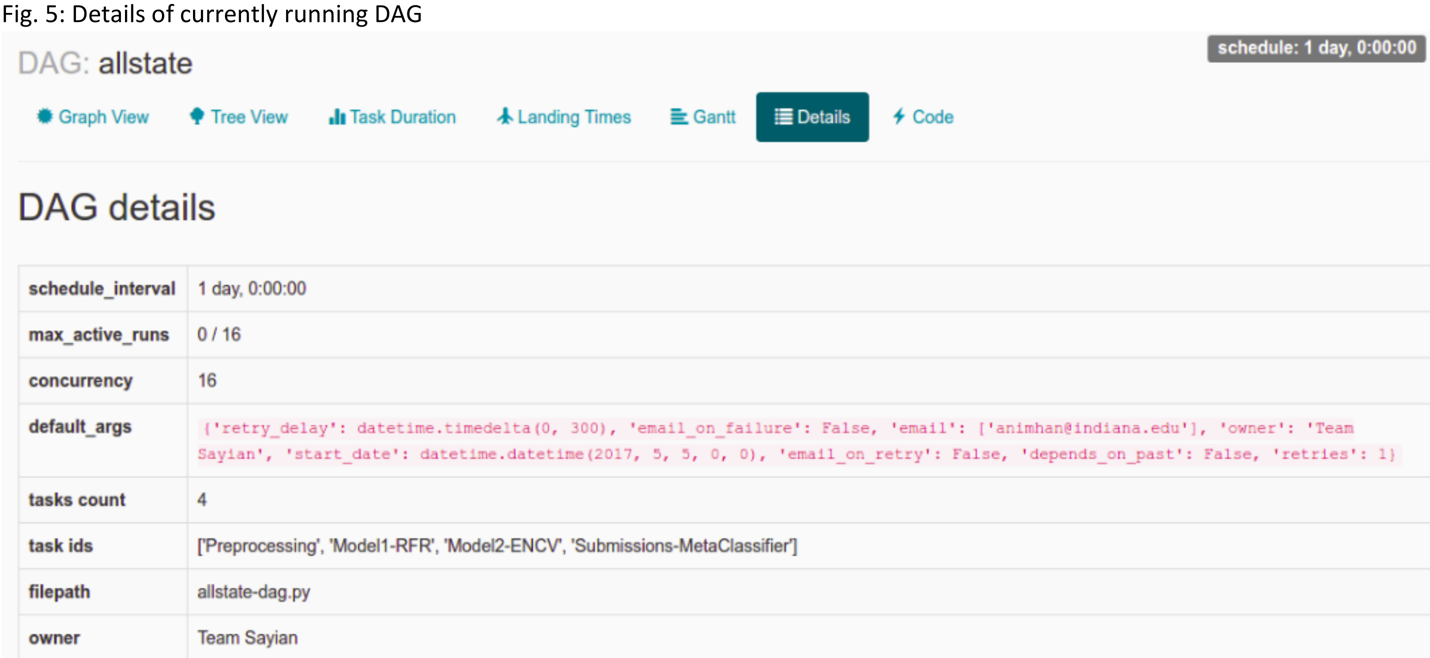


Fig. 6: Tree View of currently running DAG along with status of each task

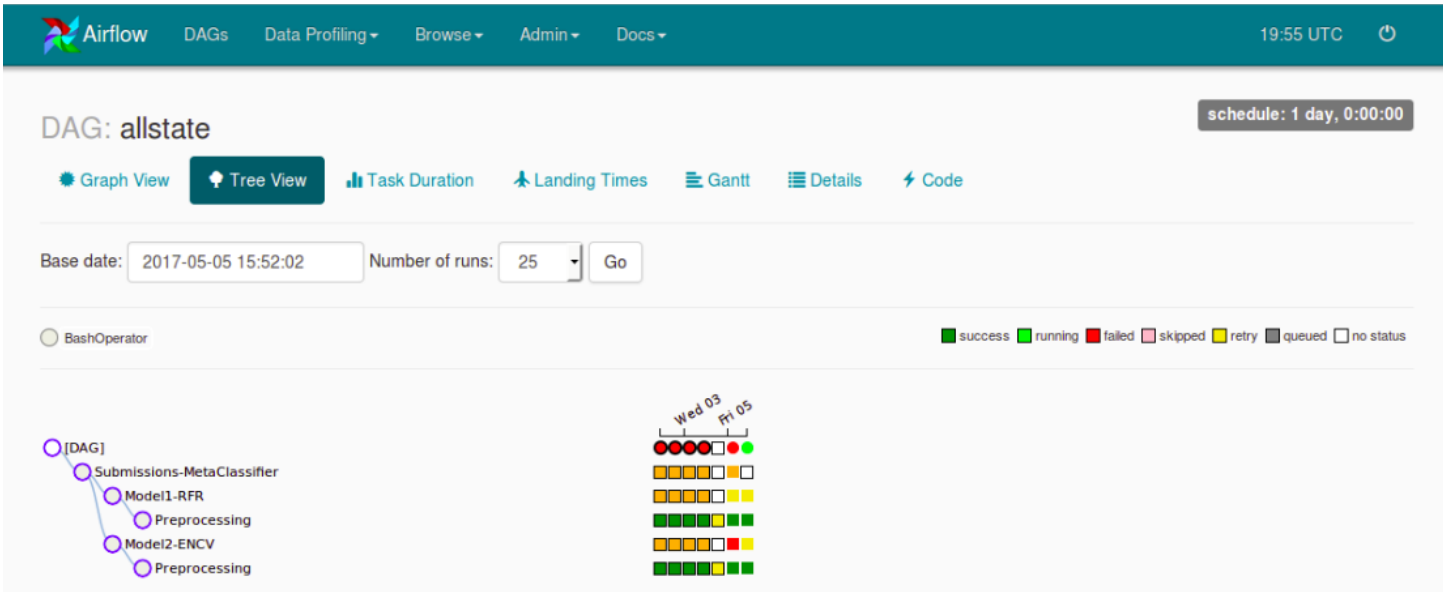


Fig. 7: Status of all tasks in current DAG along with dependencies