Overview

Deploying machine learning algorithms into an IT production environment is a challenging task to undertake due to the inherit complexity in setting up an iterative cyclic pipeline. In this paper, a novel approach for for deploying ML applications is presented; loosely based on the Cross-Industry Standard Process for Data Mining (CRISP-DM) as shown in Figure 1.

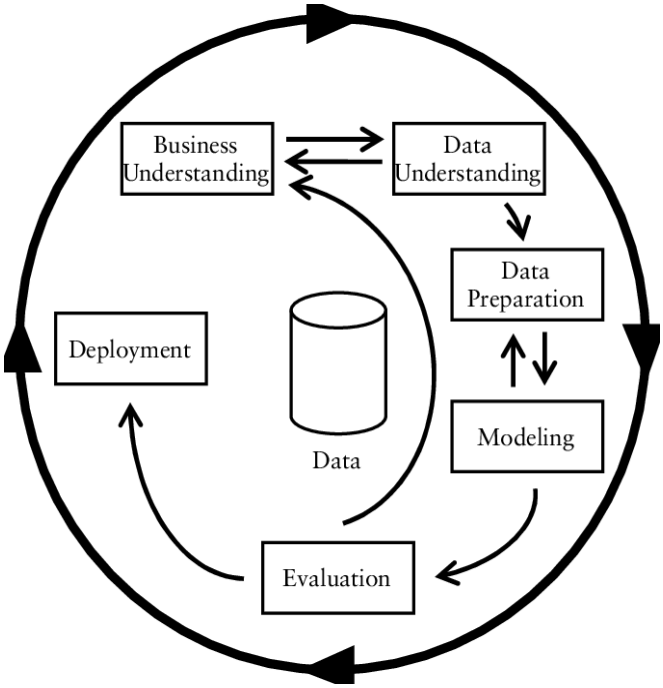


Illustration 1: CRISP-DM

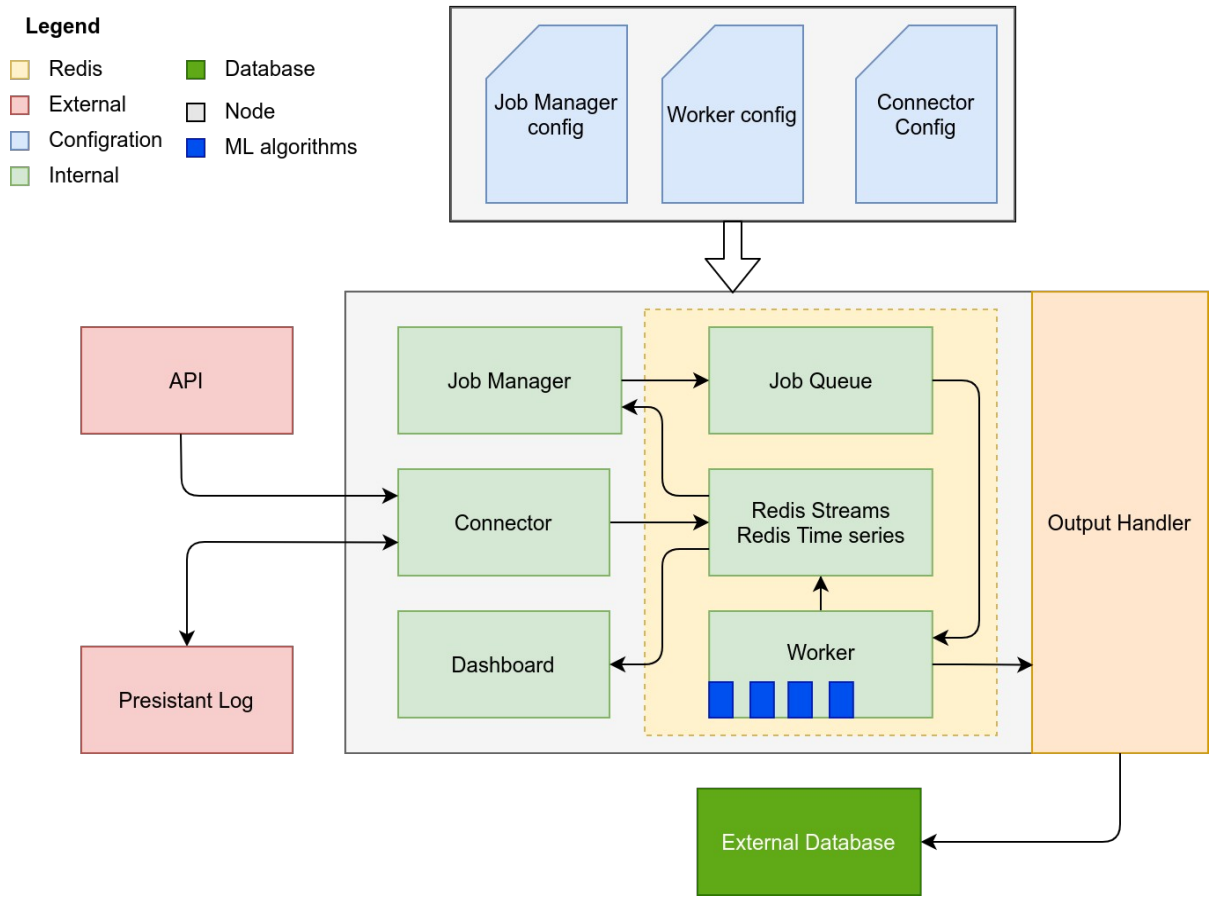
The main goal is to simplify the deployment process by imagining a simple architecture that would allow for continuous integration and deployment of ML applications; and allowing separate teams to collaborate efficiently by decoupling system dependencies using containerized solutions (Kubernetes / Docker).

Approach

In order to adhere to an agile solution, only readily available resources and systems will be used; listed below.

Requirment	Tool
Containerization	Docker
In Memory Database	Redis
Scalability	Kubernetes

Architecture



Connector: Pulls batch data from an API, inserts only new records to the streamed

Job Manager: Compares the event time of the last processed record and new records, if a time delta condition is matched, a new job is deployed

Job Queue: A simple redis job queue that holds information regarding jobs that need to be deployed

Worker: a set of workers with special ML functions that can monitor the job queue to process data based on the request of the job manager.

Dashboard: a containerized Bokeh (python) dashboard server

Code Documentation

Connector

The connector component is written purely in python. It connects to a dummy external (Sensibo) API using the OpenAPI standard. Passing configuration to the program is handled by the python3 Confuse Library. The code below shows the init function to the connector class.

Initialization:

```
def __init__(self):
        # get current working directory
        cwd = os.getcwd()
        logging.warn('Current Working Directory: {0}'.format(cwd))
        self.MSEC=1
        self.SEC=self.MSEC*1000
        self.MIN=60*self.SEC
        self.HOUR=60*self.MIN
        self.DAY=24*self.HOUR
        # When deploying this container, connector name must be specified as
        # an enviroment variable in order to pull the configuration
        try:
            if 'CONNECTORNAME' in os.environ:
                logging.info('Connector name: {0}'.format(os.environ['CONNECTORNAME']))
                self.appname=os.environ['CONNECTORNAME']
                self.config=confuse.LazyConfig(self.appname)
                logging.warn('Configuration enviroment loaded\nDetected Keys : {0}'.format(self.config.keys()))
                if self.config.keys()==[]:
                    logging.warn('No keys detected, please ensure that a configmap is configured.')
                    self._debug_config()
            else:
                raise ValueError('Enviroment variable not set: CONNECTORNAME')
        except ValueError as e:
            exit(e)
        # Log File Detection
        filename=self.config['LOG_DIR'].get()
        self.key=self.config['DATABASE_KEY'].get()
        if os.path.exists(filename):
            logging.warn('Log File Detected!')
        # Initialize the API client
        self.api_instance=self._init_api()
        # Intialize Redis Time Series Client
        self.rts=self._init_rts()
        # Create a key if it doesnt exists
        try:
            self.rts.create(self.key,retention_msecs=7*self.DAY)
        except ResponseError as e:
            s = str(e)
            logging.warning(s)
            pass
```

First the current working directory for the connector is printed out in order to help in debugging the program once deployed in a kubernetes container. Second, some constants are defined for defining time values for the redis time series structure.

The connector/application name is then queried in the environment variables. This variable is critical as it sets the directory for the configuration file. [*./config/appname*].

Note: when mounting a configmap using kubernetes, the directories must match.

Next, configuration variables are loaded such as the log file directory and redis database key that data will be streamed to. Finally the API client is initialized and a redis connection is established.

Pulling Data:

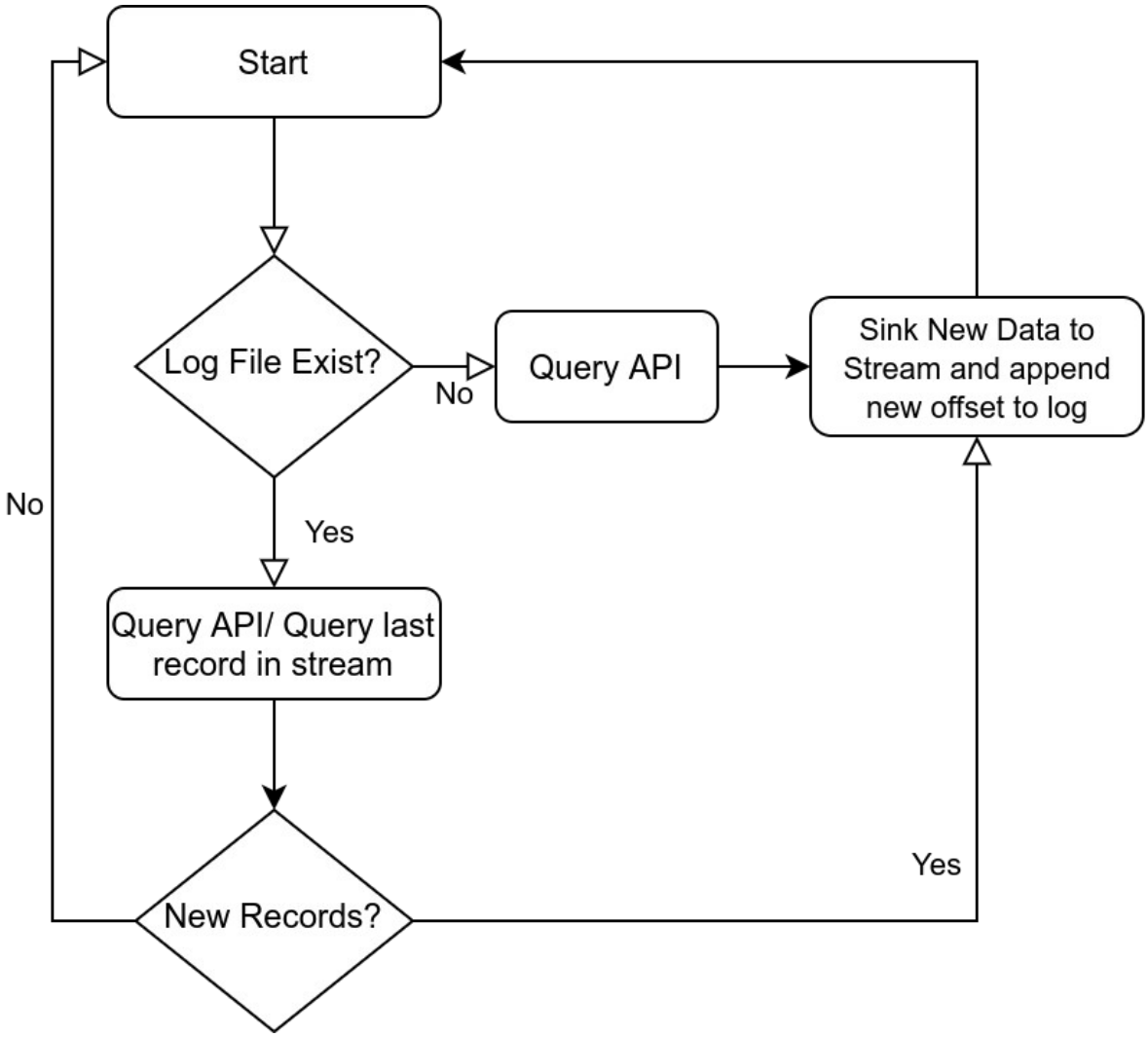
```
    def pull(self):
        try:
            HistoricalData, _, _=self.api_instance.get_pods_hist_with_http_info('MK3RxMtR',
            days=self.config['days'].get(int))
            temperature=HistoricalData.result.temperature
            parse_time=lambda datetime_str: int(mktime(datetime.strptime(datetime_str,"%Y-%m-%dT
%H:%M:%SZ" ).timetuple()))
            temperature_value,temperature_time=zip(*[[x.value,parse_time(x.time)] for x in temperature])

        except:
            logging.warn('----------------------Connection Error----------------------')
            exit('CRITICAL ERROR')
        finally:
            return temperature_time,temperature_value
```

In this application, the data consists of time series data. We query the API and return a tuple that contains two values, Time (unix epoch) and value (float). This element can be modified to accommodate connecting to different systems other than REST APIs.

Pushing Data

```
def push(self):
    stream=self.config['STREAM'].get()
    filename=self.config['LOG_DIR'].get()
    if os.path.exists(filename):
        with open(filename,'r') as log:
            # open log file and grab the id of the last messege
            last_line = log.readlines()[-1]
            # query and grab the last timestamp and parse it
            last_record=con.rts.xrevrange(stream,min='-',max=last_line,count=1)
            if last_record==[]:
                logging.critical('log file exists; last id cannot be recoverd from stream.')
                exit('critical ERROR')
            last_time=last_record[0][1]['time']
            # pull new data
            time,temp=con.pull()
            #compare new data with previously consumed timestamp
            idx = [i for i,x in enumerate(time) if x>int(last_time)]
            # if the new data has new samples parse it and update the log file
            if idx!=[]:
                new_time=time[idx[0]::]
                new_temp=temp[idx[0]::]
                ID=[con.rts.xadd(stream, dict(temp=x,time=y)) for x,y in zip(new_temp,new_time)]
                out=[(self.key,time,value) for time,value in zip(time,temp)]
                self.rts.madd(out)
                with open(filename, 'a') as log:
                    log.write(os.linesep+ID[-1])
                    logging.warn('Records added: {0}'.format(len(ID)))
            else:
                    logging.warn('No new records')
    # if the log file is missing, create a new one and pull data
    else:
        logging.warn('Could not locate log file, pulling data ...')
        os.makedirs(os.path.dirname(filename), exist_ok=True)
        time,temp =con.pull()
        ID=[con.rts.xadd(stream, dict(temp=x,time=y)) for x,y in zip(temp,time)]
        out=[(self.key,time,value) for time,value in zip(time,temp)]
        self.rts.madd(out)
        with open(filename, 'w') as log:
            log.write(ID[-1])
```

In order to push the data to redis, a the following steps are taken to maintain the offset.



This logic is reflected by the push function in the previous page.

In case a log file exists and no data is available in the stream. A critical error is raised. This could mean that the endpoints are not setup correctly or the stream has been corrupted. Furthermore, the data is pushed both to redis streams as and redis time series struct in order to both maintain the state and the ability to query the timeseries efficiently for later use cases.

Main function

The program is scheduled run every two minutes (or any configurable value) as shown below.

```
if __name__=="__main__":
    con=connector()
    import schedule
    import time
    schedule.every(2).minutes.do(con.push)
    while True:
        schedule.run_pending()
        time.sleep(1)
```

Job manager

```
def start(self,start_time):
        self.start_time=self.config['START_TIME'].get()
        self.stream=self.config['STREAM'].get()
        self.stream_log_filename=self.config['STREAM_LOG_DIR'].get()
        self.proc_log_filename=self.config['PROC_LOG_DIR'].get()
        if os.path.exists(self.stream_log_filename) & os.path.exists(self.proc_log_filename):
            with open(self.stream_log_filename,'r') as stream_log,open(self.proc_log_filename,'r') as proc_log:
                # read and update stream pointer and record and read stream
                self.stream_pointer=stream_log.readlines()[-1]
                self.new_records=self.rts.xread({self.stream: self.stream_pointer},100,1*self.SEC)
                if self.new_records!=[]:
                    # grab proc record and pointer
                    self.proc_pointer=proc_log.readlines()[-1]
                    self.proc_record=self.rts.xrange(self.stream,min=self.proc_pointer,max=self.proc_pointer,count=1)
                    self.proc_record=self.proc_record[0]
                    self.stream_record=self.new_records[0][1][-1]
                    self.stream_pointer=self.stream_record[0]
                    # log how many new records were read from the stream
                    n_records=len(self.new_records[0][1])
                    # save the stream pointer
                    with open(self.stream_log_filename, 'a') as log:
                        log.write(os.linesep+self.stream_pointer)
                        logging.warn('Records added: {0}'.format(n_records))
                    # invoke the job creation
                    self.create(self.frequency)
                else:
                    logging.warning('No New Records')

        else:
            logging.warning('Could not locate log files, scaning all available \
                data in the stream and deploying redis jobs from the beginning of the stream')
            # os.makedirs(os.path.dirname(filename), exist_ok=True)
            # consume the stream from the beggining
            self.new_records=self.rts.xread({self.stream: '0-0'},15,0)
            #### drop them into the redis time sereis!##################
            # grab the data from the latest record
            self.stream_record=self.new_records[0][1][-1]
            # grab the data from the earliest record
            self.proc_record=self.new_records[0][1][0]
            # create a pointer for the end of the stream
            self.stream_pointer=self.stream_record[0]
            #create a pointer for the jobs at the beggining of the stream
            self.proc_pointer=self.proc_record[0]
            # open the log file
            with open(self.stream_log_filename,'w') as stream_log,open(self.proc_log_filename,'w') as proc_log:
                # save the location of the pointer processing
                stream_log.write(self.stream_pointer)
                proc_log.write(self.proc_pointer)
            # invoke the job creation function
            self.create(self.frequency)
```

The job manager has two main functions:

1. Deploy jobs if new records are detected in the stream

2. Maintain the processing record

The jobs are deployed based on event time. It can be configured to deploy a job based on the time delta between events. The logic is very similar to the connector with the addition of pointers that indicate when the last job was deployed and the current status of the stream.

If a log file for the stream exists. The stream is consumed and the stream pointer is updated. Then the processing log is accessed and the record that corresponds to the last processed job is pulled. The event time is compared between the last processed record and the stream location. If the time delta condition is satisfied. A job is deployed and the processing pointer is updated.

In case a processing log file doesn't exist the stream is consumed incrementally and jobs are deployed until the end of the stream. This technique ensures that in the event of failure, all jobs are re-deployed. Furthermore, the start time can be used to override the processing log and start deploying jobs at a specified time.

Notice that a job is deployed to redis queue via reference. This is done because the job manager does not contain the source code for the deployed job. This adds flexibility where multiple specialized workers can be deployed and called based on the required job via reference.

```
def create(self,k):
    # open the processing log
    with open(self.proc_log_filename,'a') as proc_log:
        # iterate through the records and create a job based on k
        for i,x in enumerate(self.new_records[0][1]):
            t=int(x[1]['time'])
            t0=int(self.proc_record[1]['time'])
            if (t-t0)>=k:
                self.deploy_job(self.start_time,t,t0)
                # update the location of the processing pointer
                self.proc_record=self.new_records[0][1][i]
                self.proc_pointer=self.proc_record[0]
                proc_log.write(os.linesep+self.proc_pointer)
def deploy_job(self,target_time,t,t0):
    ''' logic: you can stop jobs from being deployed during fail event by supply the required start time'''
    if t0>=target_time:
        job = self.q.enqueue('prophet_algorithm.job.predict',args= (self.host,self.port,t0))
        logging.warning('JOB DEPLOYED tdelta={0}, event time start: {1}'.format(t-t0,t0))
        logging.warning(job)
```

Worker

The worker is a simple Redis Queue worker. The jobs are invoked via reference. The reference represents the python file in the working directory.

```
from redis import Redis
from rq import Queue, Worker
from os import environ
redis = Redis(host=environ['REDIS_HOST'],port=environ['REDIS_PORT'])
queue = Queue(connection=redis)
worker = Worker([queue], connection=redis)
worker.work()
```