DFFML : Labeled and Versioned Datasets

About me

1. Name : Sudharsana K J L
2. Github handle: sudharsana-kjl
3. University: National Institute of Technology, Tiruchirappalli
4. Time zone: IST
5. LinkedIn: https://www.linkedin.com/in/sudharsana-k-j-l-462640100/

Code contribution

- https://github.com/intel/dffml/pull/17
- https://github.com/intel/dffml/pull/23
- https://github.com/intel/dffml/pull/31
- https://github.com/intel/dffml/pull/38

Project information

1. Sub-org name : Data Flow Facilitator for Machine Learning (DFFML)
2. Project Abstract: Add a feature to accept Labeled and Versioned Datasets and store the data source repos in a database
3. Detailed description:

    The project is divided into two phases:

    Phase 1 : Labeling and versioning of dataset
    Phase 2: Storing the data in a database

Phase 1 : Labeling and versioning of dataset

DFFML takes the data from the dataset in an abstract source class which is used to store the data and use it in models. As of the current release, the source class doesn't accept any versioning of data. The proposed solution adds

the feature to label and store the versioned dataset.The current data is collected and stored in a Source structure like this:

```
{
    “0” : {
        "classification": "1",
        "extra": {},
        "features": {
            "PetalLength": 4.2,
            "PetalWidth": 1.5,
            "SepalLength": 5.9,
            "SepalWidth": 3.0
        },
        "last_updated": "2019-03-11T09:11:25Z",
        "prediction": {
            "classification": "1",
            "confidence": 1.0
        },
        "src_url": "0"
    },

...
}
```

This doesn’t allow us to add datasets from multiple sources that can be stored in the same JSON array in the format below:

```
{
    "default": {
        "v0": {
                        "0" : {
                           "classification": "1",
                           "extra": {},
                           "features": {
                               "PetalLength": 4.2,
                               "PetalWidth": 1.5,
                               "SepalLength": 5.9,
                               "SepalWidth": 3.0
                            },
                           "last_updated": "2019-03-11T09:11:25Z",
                           "prediction": {
                               "classification": "1",
                               "confidence": 1.0
                            },
                            "src_url": "0"
```

```
                    },
                    . . .
                }
            }
    }
}
```

Here version is **default** and label is **v0.**

Phase 2: Storing the data in a database

In the current release, the data is stored in a model directory as a cache file at a location similar to this:

```
/home/sudharsana/.cache/dffml.json
```

The model directory can be changed by using -model_dir params but still it uses file storage to store data. Since file storage is not good enough for storing huge data and analysing it, in this feature, a database is implemented in memory.py that saves the source in the database and fetches the data from the database when needed. I have analysed two approaches to solve this:

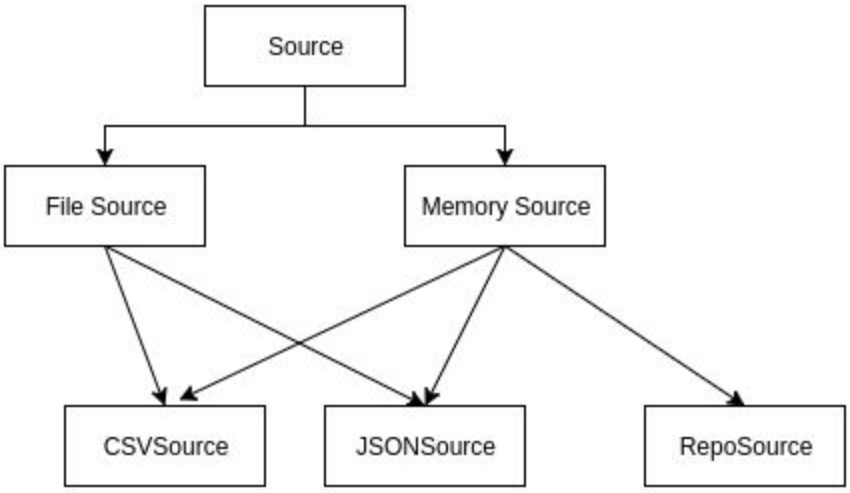
1. Using a relational database such as **SQLite**
2. Using a noSQL database like **MongoDB**
3. Using a distributed database like **Hadoop**

The detailed modifications have been discussed in the sections below.

Detailed Workflow

Phase 1 : Labeling and versioning of dataset

The hierarchy of Source class and its dependants are as follows:



In phase 1, the changes are made only for FileSource and its descendants. The outcome in phase 1 is to make the application accept labeled and versioned datasets and be able to load and dump such data in both CSV and JSON format. Phase 1 doesn't include modifying MemorySource.

In phase 1, initializing label and version in Source class is done. This is followed by modifying the helper functions in Source class to load and fetch data along with label and version. FileSource class' descendants have to be modified so that the change is transcended to the derivative classes as well. In CSVSource and JSONSource classes, the load and dump functions are updated to load and dump the data along with the label and versioning.

Phase 2: Storing the data in a database

Storing the data in a database gives a control over large set of data and aids in managing a large volume of a variety of data in one place. There are three approaches of integrating the data into a database.

1. Using a relational database like **SQLite.**

Create a SQLite database in Memory.py and create the source table when the class object is initialized. In update(), if the data already exists, it should be updated. Otherwise, data has to be inserted. In repos(), all the repos from the database which belong to a

particular label and version are returned. In repo(), a particular repo, with a specific label, version and source_url is returned.

Problem with relational databases:

Here, there is a predefined structure for features, prediction etc., that are very dataset and model dependant. Using a NoSQL database is more flexible for such a data flow facilitator.

2. Using a noSQL database like **MongoDB.**

PyMongo, a python distribution containing tools for MongoDB can be used here. Firstly a database is created in __init__ of Memory.py. The data is stored as documents that belong to a particular collection of sources that are labeled. Similar to the above approach, update(), repos() and repo() functions are modified to load and fetch the data.

Problem with non - distributed databases:

Storage of big data(huge volume of a variety of data that is changed with a velocity) used in Machine Learning applications becomes harder dude to the limitations of non distributed databases such as the database is limited to one system which leads to a single point of failure, reading and Writing of huge datasets becomes a tedious process. To avoid such issues, distributed database approach has been explored.

3. Using a distributed file storage system like **Hadoop.**

Requirement: This approach works if the user already has access to a hadoop cluster

Using hdfs, a connection can be established to the WEBHDFS URI which is a web based interface to interact with hadoop clusters. An InsecureClient (or SecureClient, if Kerberos Authentication is needed for sensitive data) connection is set up. The data from the training model is stored inside a particular file in a path like this:

http://namenodedns:port/user/hdfs/dffml/dffml.json

WEBHDFS URI path : http://namenodedns:port/user/hdfs/folder/file.csv

Default port : 5007

The data can be read and written into HDFS using read() and write() function of the client connection object.

Why this approach is more suited?

Nowadays, most of the researchers work with huge data that is accessed via hadoop. Hadoop has advantages like reliability and better response which makes it more efficient for big data storage. In this approach, DFFML can be directly used from anywhere to

connect to the cluster, run machine learning algorithms on the data and see the results. Here, huge amount of data can be trained and stored within the cluster itself which makes it more secure and also efficient for storing huge amount of data.

Scope for future enhancement:

In future, we can add support to store the data in a distributed database such as **Hive**

Expected Modifications in the code

[NOTE] *The code changes are included just to give an idea of how I would approach this problem.*

Phase 1 : Labeling and versioning of dataset

To implement this feature, we can approach like this:

1. Modify Source class to initialize two more arguments, label and version.

dffml/dffml/source/source.py:

```
class Source(abc.ABC, Entrypoint):
    '''
    Abstract base class for all sources. New sources must be derived from this
    class and implement the repos method.
    '''

    ENTRY_POINT = 'dffml.source'
    # label is set to "default" and version is set to "v0" by default
    def __init__(self, src: str, label="default", version="v0") -> None:
        self.src = src
        self.label = label
        self.version = version
```

2. Modify Source class load_from_dict function to instantiate the data in the required format with label and version

dffml/dffml/source/source.py:

```
@classmethod
```

```
    def load_from_dict(cls, sources: Dict[str, str]):
        '''
        Loads each source requested and instantiates it with its src_url.
        '''
        # Loads each source requested in the required format
        loaded: Dict[str,Dict[str,Dict[str, Source]]] = {}
        for src_url, name in sources.items():
            loaded = { self.label:
                 {self.version: { src_url:cls.load(name)(src_url)}}}
        return loaded
```

3. Modify CSVSource to add extra headers label and version if it already exists in the data being loaded.

dffml/dffml/source/csvfile.py:

```
class CSVSource(FileSource, MemorySource):
    '''
    Uses a CSV file as the source of repo feature data
    '''

    # Headers we've added to track data other than feature data for a repo
    # Label and verison headers have been added
    CSV_HEADERS = ['prediction', 'confidence', 'classification', 'label','version']

    async def load_fd(self, fd):
        '''
        Parses a CSV stream into Repo instances
        '''
        i = 0
        self.mem = {}
        for data in csv.DictReader(fd, dialect='strip'):
            # Repo data we are going to parse from this row (must include
            # features).
            repo_data = {'features': {}}
            # Parse headers we as the CSV source added
            csv_meta = {}
            for header in self.CSV_HEADERS:
                if not data.get(header) is None and data[header] != '':
                    csv_meta[header] = data[header]
                    # Remove from feature data
                    del data[header]
            # Parse feature data
            for key, value in data.items():
                if value != '':
```

```
                    try:
                        repo_data['features'][key] = ast.literal_eval(value)
                    except (SyntaxError, ValueError):
                        repo_data['features'][key] = value
            # Correct types and structure of repo data from csv_meta
            if 'classification' in csv_meta:
                repo_data.update({'classification':
                                  str(csv_meta['classification'])
                                 })
            if 'prediction' in csv_meta and 'confidence' in csv_meta:
                repo_data.update({'prediction': {
                                  'classification': str(csv_meta['prediction']),
                                  'confidence': float(csv_meta['confidence'])
                                 }})
            #Update the label and version for each repo
            if 'label' in csv_meta and 'version' in csv_meta:
                self.label = str(csv_meta['label'])
                self.version = str(csv_meta['version'])
            # Create the repo with the source URL being the row index
            repo = Repo(str(i), data=repo_data)
            i += 1
            self.mem[repo.src_url] = repo
        LOGGER.debug('%r loaded %d records', self, len(self.mem))
```

4. Modify dump_fd in CSVSource to dump the data along with label and version of the dataset.

dffml/dffml/source/csvfile.py:

```
async def dump_fd(self, fd):
        '''
        Dumps data into a CSV stream
        '''
        # Sample some headers without iterating all the way through
        fieldnames = []
        for repo in self.mem.values():
            fieldnames = list(repo.data.features.keys())
            break
        # Add our headers
        fieldnames += self.CSV_HEADERS
        # Write out the file
        writer = csv.DictWriter(fd, fieldnames=fieldnames)
        writer.writeheader()
        # Write out rows in order
        for repo in self.mem.values():
            repo_data = repo.dict()
```

```
            row = {}
            for key, value in repo_data['features'].items():
                row[key] = value
            if 'classification' in repo_data:
                row['classification'] = repo_data['classification']
            if 'prediction' in repo_data:
                row['prediction'] = repo_data['prediction']['classification']
                row['confidence'] = repo_data['prediction']['confidence']
            # Add label and version of the dataset for each repo
            row['label'] = self.label
            row['version'] = self.version
            writer.writerow(row)
        LOGGER.debug('%r saved %d records', self, len(self.mem))
```

5. Modify JSONSource to load the data and add the version and label.
dffml/dffml/source/json.py:

```
async def load_fd(self, fd):
        repos = json.load(fd)
        self.mem = {self.label : self.version: {src_url: Repo(src_url, data=data) \
                for src_url, data in repos.items()} }
        LOGGER.debug('%r loaded %d records', self, len(self.mem))
```

6. Modify JSONSource to dump the data and add the version and label.
dffml/dffml/source/json.py:

```
async def dump_fd(self, fd):
        json.dump({self.label : self.version: {repo.src_url: repo.dict()
              for repo in self.mem.values()}, fd)}
        LOGGER.debug('%r saved %d records', self, len(self.mem))
```

Phase 2: Storing the data in a database

Approach 1 : Using a relational database like SQLite

1. Set up the database and create corresponding tables

dffml/dffml/source/memory.py:

```
# SPDX-License-Identifier: MIT
# Copyright (c) 2019 Intel Corporation
'''
Fake data sources used for testing
'''
import asyncio
from typing import Dict, AsyncIterator
import sqlite3

from ..repo import Repo
from .source import Source

class MemorySource(Source):
    '''
    Stores repos in SQLite
    '''
    CREATE_SOURCE_TABLE = CREATE TABLE IF NOT EXISTS sources (
         id integer AUTOINCREMENT PRIMARY KEY,
         label text NOT NULL,
         version text NOT NULL,
         source_url integer,
         classification integer,
         extras text NOT NULL,
         last_updated text,
         features.PetalLength integer,
         features.PetalWidth integer,
         features.SepalLength integer,
         features.SepalWidth integer,
         prediction.classification integer,
         prediction.confidence integer,
    );
```

2. Add helper functions to open and close a connection to database

dffml/dffml/source/memory.py:

```
def connect(sqlite_file):
        '''
```

```
        Make connection to an SQLite database file
        '''
        conn = sqlite3.connect(sqlite_file)
        c = conn.cursor()
        return conn, c

    def close(conn):
        '''
        Commit changes and close connection to the database
        '''
        conn.commit()
        conn.close()
```

3. Modify the helper function update() to update repos in database **dffml/dffml/source/memory.py:**

```
async def update(self, repo):
    '''
        UPSERT a repo along with label and version
        '''
        update_sources_sql = UPDATE sources \
                             SET label = ? ,\
                               version = ? ,\
                               source_url = ? ,\
                               classification = ? ,\
                               extras = ? ,\
                               last_updated = ? ,\
                               features_PetalLength = ? ,\
                               features_PetalWidth = ? ,\
                               features_SepalLength = ? ,\
                               features_SepalWidth = ? ,\
                               prediction_classification = ? ,\
                               prediction_confidence = ?
                              WHERE id = ?

        insert_sources_sql = INSERT OR IGNORE INTO sources \
                             (label, version, source_url, classification, extras,\
                                last_updated, features_PetalLength, \
                                features_PetalWidth, features_SepalLength,\
                                  features_SepalWidth, prediction_classification,\
                                   prediction_confidence) VALUES \
                                    ( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

        async with self.lock:
```

```
            conn, c = connect(self.sqlite_file)
            get_id_sql = SELECT id from sources WHERE \
                            label=self.label, version=self.version, \
                             source_url=repo.src_url
            c.execute(get_id_sql)
            object_id = c.fetchone()

            # Try to update an existing repo
            c.execute(update_sources_sql,(self.label,self.version,\
                       repo.src_url,repo.classification,\
                       JSON.stringify(repo.extras), repo.last_updated,\
                       repo.features.PetalLength, repo.features.PetalWidth,\
                       repo.features.SepalLength, repo.features.SepalWidth,\
                       repo.prediction.classification, \
                       repo.prediction.confidence, object_id))
            # Try to insert if it doesn't exist
            c.execute(insert_sources_sql,((self.label,self.version,\
                       repo.src_url,repo.classification,\
                       JSON.stringify(repo.extras), repo.last_updated,\
                       repo.features.PetalLength, repo.features.PetalWidth,\
                       repo.features.SepalLength, repo.features.SepalWidth,\
                       repo.prediction.classification, \
                       repo.prediction.confidence, object_id)))
            close(conn)
```

4. Modify the function repos() to fetch all data from the database **dffml/dffml/source/memory.py:**

```
async def repos(self) -> AsyncIterator[Repo]:
        # NOTE No lock used here because sometimes we iterate and update
        # Feel free to debate this by opening an issue.
        get_all_repos_sql = SELECT * from sources
        conn, c = connect(self.sqlite_file)
        c.execute(get_all_repos_sql)
        values = c.fetchall()

        for repo in values:
            ## TODO Convert to Repo object JSON format
            yield repo
        close(conn)
```

5. Modify the function repo() to get a specific repo **dffml/dffml/source/memory.py:**

```
async def repo(self, src_url: str, label: str, version: str) -> Repo:
        async with self.lock:
            get_specific_repo_sql = SELECT * from sources WHERE label=label, \
                     version=version, source_url=src_url
            conn, c = connect(self.sqlite_file)
            c.execute(get_specific_repo_sql)
            value = c.fetchone()
            ## TODO Convert to Repo object JSON format
            close(conn)
            return value
```

Approach 2 : Using a NoSQL database like MongoDB

1. Set up the database and create corresponding collections

dffml/dffml/source/memory.py:

```
# SPDX-License-Identifier: MIT
# Copyright (c) 2019 Intel Corporation
'''
Fake data sources used for testing
'''
import asyncio
from typing import Dict, AsyncIterator
from pymongo import MongoClient

from ..repo import Repo
from .source import Source

class MemorySource(Source):
    '''
    Stores repos in a dict in memory
    '''

    def __init__(self, src: str) -> None:
        super().__init__(src)
        client = MongoClient(<<MONGODB URL>>)
        # Create a database object
        db= client.dffml_source
        self.lock = asyncio.Lock()
```

2. Modify the helper function update() to update repos in database

dffml/dffml/source/memory.py:

```
async def update(self, repo):
        async with self.lock:
            db.sources.update_one({'label': self.label, 'version' : self.version,
'src_url' : repo.src_url}, repo)
```

3. Modify the function repos() to fetch all data from the database

dffml/dffml/source/memory.py:

```
async def repos(self) -> AsyncIterator[Repo]:
        # NOTE No lock used here because sometimes we iterate and update
        # Feel free to debate this by opening an issue.
        for repo in db.sources.find({}, {'_id': false}):
            yield repo
```

4. Modify the function repo() to get a specific repo given a source_url, label and version. Also modify RepoSource to write data in the required format.

dffml/dffml/source/memory.py:

```
async def repo(self, src_url: str) -> Repo:
        async with self.lock:
            return db.sources.find_one({'label': self.label, 'version' :
self.version, 'src_url' : repo.src_url}, {'_id': false})

class RepoSource(MemorySource):
    '''
    Takes repo data from instantiation arguments. Stores repos in memory.
    '''

    def __init__(self, *args: Repo, src: str = '') -> None:
        super().__init__(src)
        db.sources.insert_one({'label': self.label, 'version' : self.version,
'src_url' : repo.src_url,repo.src_url: repo for repo in args}
```

Approach 3 : Using distributed database like Hadoop

1. Import dependent libraries and set up a client connection.

dffml/dffml/source/memory.py:

```
# SPDX-License-Identifier: MIT
# Copyright (c) 2019 Intel Corporation
'''
Fake data sources used for testing
'''
import asyncio
from typing import Dict, AsyncIterator
from hdfs import InsecureClient
import os

from ..repo import Repo
from .source import Source

class MemorySource(Source):
    '''
    Stores repos in a dict in HDFS
    '''

    def __init__(self, src: str) -> None:
        super().__init__(src)
        # Connecting to WebHDFS by providing HDFS host IP and WebHDFS port (50070
by default)
        self.client_hdfs = InsecureClient('http://' + os.environ['IP_HDFS'] +
':50070')
        self.file_path = '/user/hdfs/dffml/dffml.json'
        self.lock = asyncio.Lock()
```

2. Modify update() to update repos in HDFS

dffml/dffml/source/memory.py:

```
async def update(self, repo):
        async with self.lock:
            repos = None
            # There is no Update functionality in HDFS
            # Reading data from hdfs
            with self.client_hdfs.read(self.file_path) as reader:
                repos = reader.read()
            # Update repo
            repos[self.label][self.version][repo.src_url] = repo
            # The existing file may have to be deleted before writing since HDFS
follows WORM principle
            # Writing updated data into hdfs
```

```
            with self.client_hdfs.write(self.file_path) as writer:
                writer.write(repos)
```

3. Modify the function repos() to fetch all data from HDFS

dffml/dffml/source/memory.py:

```
async def repos(self) -> AsyncIterator[Repo]:
        # NOTE No lock used here because sometimes we iterate and update
        # Feel free to debate this by opening an issue.
        with self.client_hdfs.read(self.file_path) as reader:
            content = reader.read()
            yield content[self.label][self.version]
```

4. Modify the function repo() to get a specific repo given a source_url, label and version

dffml/dffml/source/memory.py:

```
async def repo(self, src_url: str) -> Repo:
        async with self.lock:
            with self.client_hdfs.read(self.file_path) as reader:
                repos = reader.read()
                return repos[self.label][self.version][src_url]
```

5. Modify RepoSource to write data into HDFS

dffml/dffml/source/memory.py:

```
class RepoSource(MemorySource):
    '''
    Takes repo data from instantiation arguments. Stores repos in HDFS.
    '''

    def __init__(self, *args: Repo, src: str = '') -> None:
        super().__init__(src)
        self.client_hdfs.write(self.file_path, {self.label: { self.version:
{repo.src_url: repo for repo in args}})
```

Weekly timeline

- **Community Bonding (May 7-26):**

Getting to understand the current release of the sub-organisation and understanding the flow. Discussing the changes with the mentor and get his feedback

- **Week 1 (May 27-31):**

  Start coding and implement the function modifications in Source.py by having a default label and version data

- **Week 2 (June 3):**

  Make necessary modifications in FileSource and its derivatives CSVSource

- **Week 3 (June 10):**

  Modify JSONSource to include the label and version as well

- **Week 4 (June 17) & Week 5 (June 24):**

  Writing test cases for phase 1 modifications

- **Week 6 (July 1):**

  Getting phase 1 work reviewed and accomplished

- **Week 7 (July 8):**

  Start work on phase 2 by setting up the necessary database connections

- **Week 8 (July 15):**

  Loading the data into the database in a specific format

- **Week 9 (July 22):**

  Modifying helper function to update the data in the database

- **Week 10 (July 29):**

  Modifying helper function to get all the repos as well as filter them

- **Week 11 (August 5):**

Writing test cases for phase 2 connections and data insertion into the database

- **Week 12 (August 12):**

  Writing test cases for data updation and data fetch from the database

- **Final week (August 19):**

  Phase 2 review by mentor and overall final review after which project has to be submitted.

Future Enhancements

1. Creating a pipeline to get data from hadoop and perform machine learning and write the modified data back into hadoop
2. Adding support for Kafka to get real-time data
3. Using a distributed database such as **Hive** to store the data

Other commitments

- None