# Scale-independent Data Analysis with Database-backed Dataframes: a Case Study

Phanwadee Sinthong
University of California, Irvine
California, USA
psinthon@uci.edu

Yuhan Yao
University of California, Irvine
California, USA
yuhany2@uci.edu

Michael J. Carey
University of California, Irvine
California, USA
mjcarey@ics.uci.edu

## ABSTRACT

Big Data analytics are being used in many businesses and organizations. However, the requirements that big data analytics introduce cannot be solved by a single system, and require distributed system knowledge and data management expertise from data analysts who should instead be focus on gleaning information from the data. AFrame is a data exploration library that provides a data scientist's familiar interface, Pandas DataFrame, and scales its operations to large volumes of data through a big data management system to enable a seamless analysis experience moving from small to large datasets. In this paper, we present a detailed case study that uses AFrame to perform an end-to-end data analysis from data cleaning and modeling through to deployment.

## 1 INTRODUCTION

As large volumes of data are generated and available through various sources (e.g., social platforms, IoT, and daily transactions), Big Data analytics are gaining popularity across all types of industries. Harnessing information and patterns from data can help businesses identify opportunities, make well-informed decisions, and manage risks. That, in turn, leads to higher profits and satisfied customers. Even though Big Data production is evident, efficient Big Data processing and analysis remain challenging. Existing data analytic tools and libraries require an extensive knowledge of distributed data management from data analysts who should instead be focusing on tasks like developing and using machine learning models.

AFrame [13] is a data exploration and analysis library that delivers a scale-independent analysis experience by providing a Pandas-like DataFrame interface while transparently scaling the analytical operations to be executed on a database system. AFrame leverages database data storage and management in order to accommodate the rate and volume at which the data arrives. It allows data scientists to perform data analysis right where the data is stored. AFrame simplifies user interactions with large amounts of data, as the framework provides data scientists' familiar interface and operates directly on database systems without requiring multiple system setups.

In this paper, we illustrate the usability of AFrame through a case study that uses it to perform an end-to-end data analysis. We highlight some of AFrame's functionalities that help simplify Big Data analysis through each of the data analytics lifecycle stages. The notebook that we use in this paper is also available [1].

The rest of this paper is organized as follows: Section 2 discusses background and related work. Section 3 details our case study. We conclude and describe our current work in Section 4.

[1]https://nbviewer.jupyter.org/github/psinthong/SF_CRIME_Notebook/blob/master/sf_crimes_paper.ipynb

## 2 BACKGROUND AND RELATED WORK

We are developing AFrame to lighten user workloads and ease the transitioning from a local workstation to a production environment by providing data scientists with their familiar interface, Pandas DataFrames, and transparently scale its operations to execute in a distributed database environment. Here we briefly review Pandas, AFrame's architecture, and related work.

### 2.1 Pandas

Pandas [5] is an open source Python data analytic library. Pandas provides a data structure called DataFrame, designed specifically for data manipulation and analysis. DataFrames are similar to a table in Excel with labeled rows and columns. Pandas works with several popular machine learning libraries such as Scikit-Learn [11] and Tensorflow [8]. Pandas is one of the most popular data analytic libraries partly due to its flexible data structure and the rich set of features that the library provides. However, Pandas' shortcoming lies in its scalability, as it only operates on a single workstation and utilizes a single processing core.

### 2.2 AFrame

AFrame is a Python library that provides a Pandas DataFrame-like syntax to interact with distributed data stored in a big data management system, Apache AsterixDB [9]. AFrame utilizes lazy evaluation to scale Pandas operations by incrementally constructing database queries from each of Pandas DataFrame operations and it only sends the queries over to AsterixDB when results are actually called for. AFrame delivers a Pandas DataFrame-like experience on Big Data requiring minimum effort from data scientists allowing them to focus on analyzing the data.

### 2.3 Related Work

There are several scalable dataframe libraries that try to deliver a Pandas-like experience and scale operations onto large volumes of file-based data using different methods. These libraries either provide a similar Pandas-like interface on a distributed compute engine, execute several Pandas DataFrames in parallel, or use memory mapping to optimize the computation and speed up the data access. To our knowledge, there has not been any effort to develop a Pandas-like interface directly on top of database systems where large volumes of data are stored. AFrame can leverage database index and query optimizer to efficiently access and operate on data at scale without moving it into system-specific environment or create an intermediate data representation. Here we briefly compare and contrast two of the most well-known scalable dataframe libraries, Spark [10] and Dask [12].

*2.3.1 Spark.* Apache Spark [2] is a general-purpose cluster computing framework. Spark provides a DataFrame API, an interface for data analysts to interact with data in a distributed file system. However, Spark's DataFrame syntax is quite different from Pandas', as Spark is heavily influenced by SQL's syntax while Pandas relies on features of the Python language. As a result, the Koalas project [4] was introduced to bridge the gap between Spark and Pandas DataFrames. Koalas is a Python library implemented on top of the Spark DataFrame API and its syntax is designed to be largely identical to that of Pandas'. In order to support Pandas DataFrame features (e.g., row label, eager evaluation) in a distributed environment, Koalas implements an intermediate data representation that results at times in expensive operations and performance trade-offs. Since Koalas operates directly on top of Spark, users are also required to set up distributed file storage as well as tuning Spark to suit their data access patterns.

*2.3.2 Dask.* Dask is an open source Python framework that provides advanced parallelism for data scientists' familiar analytical libraries such as Pandas, NumPy, and Scikit-learn. Dask DataFrame is a scalable dataframe library that is composed of multiple Pandas DataFrames. Dask partitions its data files row-wise and operates on them in parallel while utilizing all of the available computational cores. Since Dask uses Pandas internally, it also inherits the high memory consumption problem that Pandas has. Dask tries to compensate for this disadvantage by delaying its expression evaluation in order to reduce the amount of needed computation. However, Dask does not aim at implementing all of the Pandas operations because supporting certain Pandas operations in a distributed environment results in poor performance trade-offs, as mentioned in its documentation [3].

## 3 AFRAME CASE STUDY

There are several methodologies that provide a guideline for the stages in a data science lifecycle, such as the traditional method called CRISP_DM for data mining and an emerging methodology introduced by Microsoft called TDSP [7]. They have defined general phases in a data analysis lifecycle that can be summarized as follows: business understanding, data understanding and preparation, modeling, evaluation, and deployment.

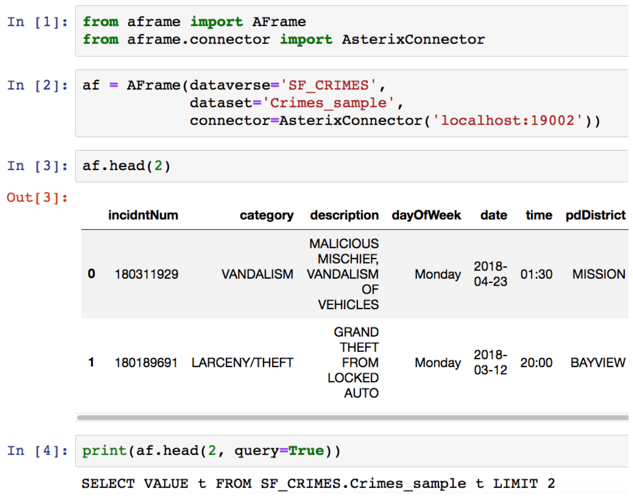
Our goal for AFrame is to deliver a scale-independent data analysis experience. Data analysts should be able to use AFrame interchangeably with Pandas DataFrames and utilize their favorite ML libraries to perform each of the data science lifecycle stages on large volumes of data with minimum effort.

In order to see if AFrame delivers up to our expectations, we conducted a case study by asking a user to perform a data analysis using AFrame in places where Pandas DataFrames would otherwise be used (with an exception of training machine learning models). We used a running example of an analysis of a San Francisco police department historical incident report dataset [6] to predict the probability of incidents being resolved. Due to space limitations, we will only be displaying a subset of the printed attributes in our Figures.

We present the case study here through each of the previously mentioned data science lifecycle stages. (Interested readers can find information about AFrame's performance in [13].)

### 3.1 Data understanding and preparation

The goal of this stage in the data science lifecycle is to obtain and clean the data to prepare it for analysis. Figure 1 is a snapshot from a Jupyter notebook that shows a process of creating an AFrame object and displaying two records from a dataset. Input line 2 labeled 'In[2]' shows how to create an AFrame object by utilizing AFrame's AsterixDB connector and providing a dataverse name and a dataset name. For this example, the dataset is called 'Crimes_sample' and it is contained in a dataverse named 'SF_CRIMES'. The AsterixDB server is running locally on port 19002. Input line 3 displays a sample of two records from the dataset, and input line 4 displays the underlying SQL++ query that AFrame generated and extended. More about AFrame's incremental query formation process can be found in [13].



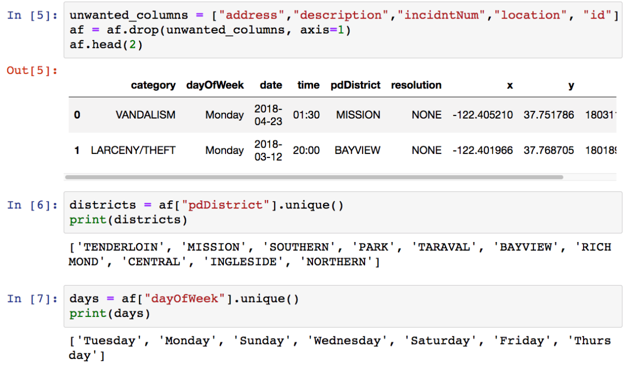
```
In [1]:  from aframe import AFrame
         from aframe.connector import AsterixConnector

In [2]:  af = AFrame(dataverse='SF_CRIMES',
                     dataset='Crimes_sample',
                     connector=AsterixConnector('localhost:19002'))

In [3]:  af.head(2)
Out[3]:
```

| | incidntNum | category | description | dayOfWeek | date | time | pdDistrict |
|---|---|---|---|---|---|---|---|
| 0 | 180311929 | VANDALISM | MALICIOUS MISCHIEF, VANDALISM OF VEHICLES | Monday | 2018-04-23 | 01:30 | MISSION |
| 1 | 180189691 | LARCENY/THEFT | GRAND THEFT FROM LOCKED AUTO | Monday | 2018-03-12 | 20:00 | BAYVIEW |

```
In [4]:  print(af.head(2, query=True))

         SELECT VALUE t FROM SF_CRIMES.Crimes_sample t LIMIT 2
```

**Figure 1: Acquire data**

Next, our user drops some columns from the dataset and explores the data values, as shown in Figure 2. Input line 5 drops several columns using the Pandas' 'drop' function and prints out two records from the resulting data. Our user then prints out the unique values from the columns 'pdDistrict' and 'dayOfWeek' as shown in input lines 6 and 7 respectively.



```
In [5]:  unwanted_columns = ["address","description","incidntNum","location", "id"]
         af = af.drop(unwanted_columns, axis=1)
         af.head(2)
Out[5]:
```

| | category | dayOfWeek | date | time | pdDistrict | resolution | x | y | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | VANDALISM | Monday | 2018-04-23 | 01:30 | MISSION | NONE | -122.405210 | 37.751786 | 18031 |
| 1 | LARCENY/THEFT | Monday | 2018-03-12 | 20:00 | BAYVIEW | NONE | -122.401966 | 37.768705 | 18018 |

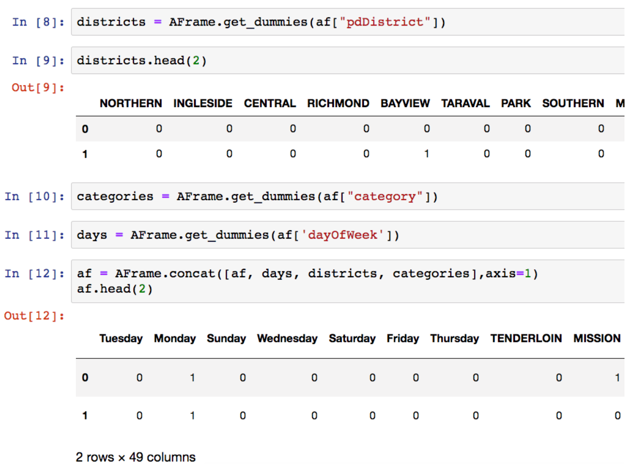
```
In [6]:  districts = af["pdDistrict"].unique()
         print(districts)

         ['TENDERLOIN', 'MISSION', 'SOUTHERN', 'PARK', 'TARAVAL', 'BAYVIEW', 'RICH
         MOND', 'CENTRAL', 'INGLESIDE', 'NORTHERN']

In [7]:  days = af["dayOfWeek"].unique()
         print(days)

         ['Tuesday', 'Monday', 'Sunday', 'Wednesday', 'Saturday', 'Friday', 'Thurs
         day']
```

**Figure 2: Data cleaning and exploration**

### 3.2 Modeling

The next stage in a data science project lifecycle is to determine and optimize features through feature engineering to facilitate machine learning model training. This stage also includes machine learning model development, which is the process to construct and select a model that can predict the target values most accurately considering their success metrics.

In Figure 3, the user applies one-hot encoding by utilizing the Pandas' 'get_dummies' function to create multiple features from the columns that he previously explored. Input line 8 applies one-hot encoding to the 'pdDistrict' column and line 9 displays the resulting data with ten new columns each indicating whether or not the record has that particular feature. Input lines 10 and 11 perform the same operation on the 'category' and 'dayOfWeek' columns respectively. The user then appends all of their one-hot encodings to the original data in input line 12.



```
In [8]:  districts = AFrame.get_dummies(af["pdDistrict"])

In [9]:  districts.head(2)
Out[9]:
```

| | NORTHERN | INGLESIDE | CENTRAL | RICHMOND | BAYVIEW | TARAVAL | PARK | SOUTHERN | M |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **1** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |

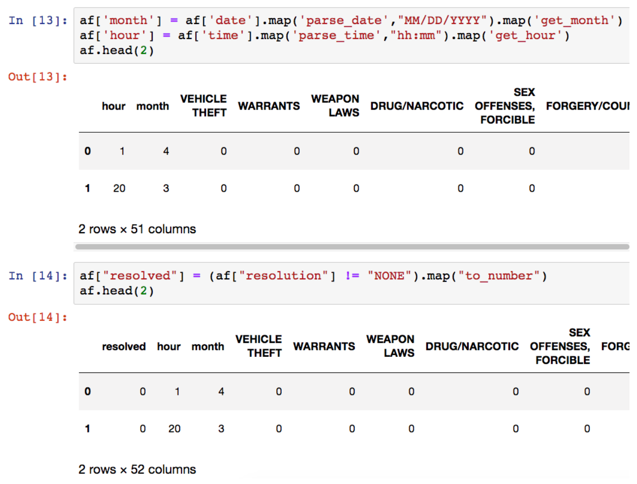
```
In [10]: categories = AFrame.get_dummies(af["category"])

In [11]: days = AFrame.get_dummies(af['dayOfWeek'])

In [12]: af = AFrame.concat([af, days, districts, categories],axis=1)
         af.head(2)
Out[12]:
```

| | Tuesday | Monday | Sunday | Wednesday | Saturday | Friday | Thursday | TENDERLOIN | MISSION |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **1** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

2 rows × 49 columns

**Figure 3: One-hot encodings**

In order to extract important features from the data, users can also apply AsterixDB's builtin functions directly on the entire data or part of the data. This is done through the 'map' and 'apply' functions. The complete list of all available builtin functions can be found at [1]. Figure 4 shows an example of using the map operation on a subset of the data attributes. Input line 13 creates two new columns, 'month' and 'hour'. For 'month', the user applies AsterixDB's 'parse_date' to the 'date' column to generate a date object and then applies the 'get_month' function to extract only the month before appending it as a new column called 'month'. Similarly, for the 'hour' column, 'parse_time' is applied to the 'time' column followed by the 'get_hour' function to extract the hour of day from the data before appending it as a new column. Finally, to finish up the feature engineering process, the target column 'resolution' is converted into a binary value column called 'resolved' using AsterixDB's 'to_number' function.



```
In [13]: af['month'] = af['date'].map('parse_date',"MM/DD/YYYY").map('get_month')
         af['hour'] = af['time'].map('parse_time',"hh:mm").map('get_hour')
         af.head(2)
Out[13]:
```

| | hour | month | VEHICLE THEFT | WARRANTS | WEAPON LAWS | DRUG/NARCOTIC | SEX OFFENSES, FORCIBLE | FORGERY/COUI |
|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 4 | 0 | 0 | 0 | 0 | 0 | |
| **1** | 20 | 3 | 0 | 0 | 0 | 0 | 0 | |

2 rows × 51 columns

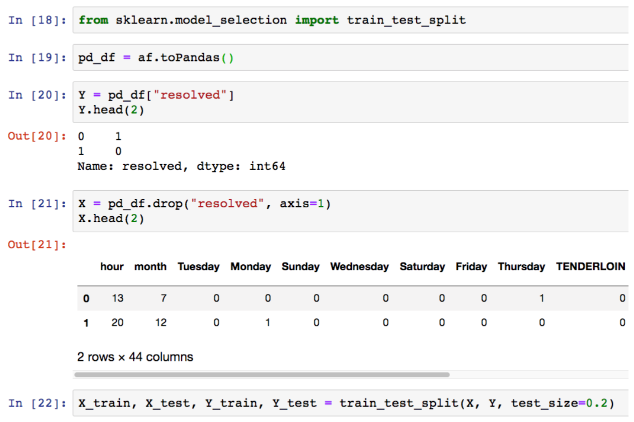
```
In [14]: af["resolved"] = (af["resolution"] != "NONE").map("to_number")
         af.head(2)
Out[14]:
```

| | resolved | hour | month | VEHICLE THEFT | WARRANTS | WEAPON LAWS | DRUG/NARCOTIC | SEX OFFENSES, FORCIBLE | FORG |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 4 | 0 | 0 | 0 | 0 | 0 | |
| **1** | 0 | 20 | 3 | 0 | 0 | 0 | 0 | 0 | |

2 rows × 52 columns

**Figure 4: Applying functions to create new columns**

Once the feature engineering process is done, the data is split into training and testing sets for use in training and evaluating machine learning models. Figure 5 shows the process of splitting the data. Input line 19 converts the data referenced by an AFrame object into a Pandas DataFrame. Currently, feeding data into existing Scikit-learn models from a database system is not supported. As such, AFrame provides an operation called 'toPandas' which converts data into Pandas DataFrame objects[2]. On input line 20, 'Y' is the binary encoded 'resolved' column and the remaining columns will be used to train the models. Input line 22 splits the data into an 80% training set and a 20% testing set.



```
In [18]: from sklearn.model_selection import train_test_split

In [19]: pd_df = af.toPandas()

In [20]: Y = pd_df["resolved"]
         Y.head(2)
Out[20]: 0    1
         1    0
         Name: resolved, dtype: int64

In [21]: X = pd_df.drop("resolved", axis=1)
         X.head(2)
Out[21]:
```

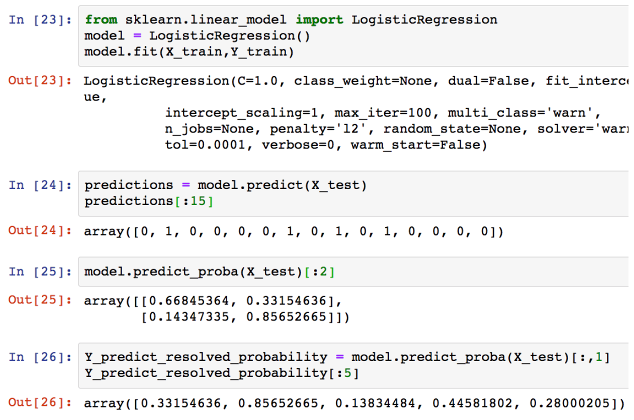
| | hour | month | Tuesday | Monday | Sunday | Wednesday | Saturday | Friday | Thursday | TENDERLOIN |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 13 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **1** | 20 | 12 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

2 rows × 44 columns

```
In [22]: X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2)
```

**Figure 5: Preparing data for model training**

Input lines 23 - 26 in Figure 6 are standard Scikit-learn model training steps that take a Pandas DataFrame as their input. Input line 23 trains a Logistic Regression model on the training data, while input line 24 calls the 'predict' function on the test data and displays a subset of the results. Instead of returning binary results indicating whether or not a particular incident will get resolved, users can utilize Scikit-learn's 'predict_proba' method to get the raw probability values that the model outputs for each of the prediction labels (0 and 1 in our case). Input line 25 shows the probability values that the model outputs for each of the labels in order (0 followed by 1) on a subset of the records. Our user decided to use the 'predict_proba' function and output the probability of an incident getting resolved as shown in line 26.



```
In [23]: from sklearn.linear_model import LogisticRegression
         model = LogisticRegression()
         model.fit(X_train,Y_train)

Out[23]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_interc
         ue,
                   intercept_scaling=1, max_iter=100, multi_class='warn',
                   n_jobs=None, penalty='l2', random_state=None, solver='war
                   tol=0.0001, verbose=0, warm_start=False)

In [24]: predictions = model.predict(X_test)
         predictions[:15]

Out[24]: array([0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0])

In [25]: model.predict_proba(X_test)[:2]

Out[25]: array([[0.66845364, 0.33154636],
                [0.14347335, 0.85652665]])

In [26]: Y_predict_resolved_probability = model.predict_proba(X_test)[:,1]
         Y_predict_resolved_probability[:5]

Out[26]: array([0.33154636, 0.85652665, 0.13834484, 0.44581802, 0.28000205])
```

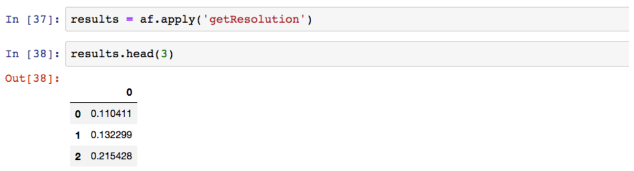
**Figure 6: Model training and inferencing**

## 3.3 Evaluation and deployment

In order to deploy the model into a production environment and apply it to large datasets at full scale, users can export and package their models using Python's pickle and then deposit them into AsterixDB for use as external user-defined functions (UDFs). In our example, the user deposited their model and created a function called 'getResolution' in AsterixDB that invokes the

[2]The resulting data is required by Pandas to fit in memory. This currently limits the training dataset size, but there is no limit to the amount of data to which the model may be applied (see Section 3.3).

trained model's 'predict_proba' function on a data record. Due to space limitations, we omit the steps to deposit the model into AsterixDB and create a new UDF to call it, but the required steps can be found in [1]. After creating a UDF, users can then utilize their model using AFrame in the same way that they would use a builtin function. Figure 7 shows the user applying their Python UDF 'getResolution' on an AFrame object. Line 37 uses the 'apply' operation to apply the Python UDF on the previously transformed data. The results of the function are the probabilities of crime incidents getting resolved, as displayed in line 38.

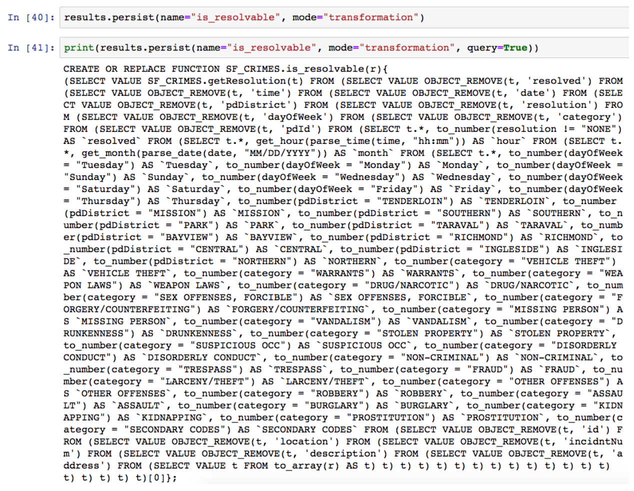


```
In [37]: results = af.apply('getResolution')

In [38]: results.head(3)
Out[38]:
```

|   | 0 |
|---|---|
| 0 | 0.110411 |
| 1 | 0.132299 |
| 2 | 0.215428 |

**Figure 7: Calling the model using the function syntax**

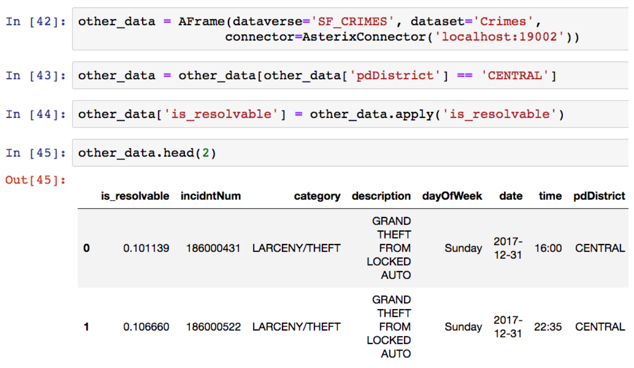
At this point in the analysis, our user is done with the training and evaluation process and wants to apply the model to other larger sets of data. However, to apply the model to a different dataset, that dataset has to have the same features arranged in the same order as the training data. To simplify the model inferencing process, AFrame allows users to save their data transformation as a function that can then be applied to other datasets effortlessly. Line 40 in Figure 8 displays the persist-transformation syntax. The user has named the resulting function 'is_resolvable'. The underlying SQL++ query that transforms and appends their engineered features to a data record before calling the trained model on it is shown in input line 41.



```
In [40]: results.persist(name="is_resolvable", mode="transformation")

In [41]: print(results.persist(name="is_resolvable", mode="transformation", query=True))

         CREATE OR REPLACE FUNCTION SF_CRIMES.is_resolvable(r){
         (SELECT VALUE SF_CRIMES.getResolution(t) FROM (SELECT VALUE OBJECT_REMOVE(t, 'resolved') FROM
         (SELECT VALUE OBJECT_REMOVE(t, 'time') FROM (SELECT VALUE OBJECT_REMOVE(t, 'date') FROM (SELE
         CT VALUE OBJECT_REMOVE(t, 'pdDistrict') FROM (SELECT VALUE OBJECT_REMOVE(t, 'resolution') FRO
         M (SELECT VALUE OBJECT_REMOVE(t, 'dayOfWeek') FROM (SELECT VALUE OBJECT_REMOVE(t, 'category')
         FROM (SELECT VALUE OBJECT_REMOVE(t, 'pdId') FROM (SELECT t.*, to_number(resolution != "NONE")
         AS `resolved` FROM (SELECT t.*, get_hour(parse_time(time, "hh:mm")) AS `hour` FROM (SELECT t.
         *, get_month(parse_date(date, "MM/DD/YYYY")) AS `month` FROM (SELECT t.*, to_number(dayOfWeek
         = "Tuesday") AS `Tuesday`, to_number(dayOfWeek = "Monday") AS `Monday`, to_number(dayOfWeek =
         "Sunday") AS `Sunday`, to_number(dayOfWeek = "Wednesday") AS `Wednesday`, to_number(dayOfWeek
         = "Saturday") AS `Saturday`, to_number(dayOfWeek = "Friday") AS `Friday`, to_number(dayOfWeek
         = "Thursday") AS `Thursday`, to_number(pdDistrict = "TENDERLOIN") AS `TENDERLOIN`, to_number
         (pdDistrict = "MISSION") AS `MISSION`, to_number(pdDistrict = "SOUTHERN") AS `SOUTHERN`, to_n
         umber(pdDistrict = "PARK") AS `PARK`, to_number(pdDistrict = "TARAVAL") AS `TARAVAL`, to_numb
         er(pdDistrict = "BAYVIEW") AS `BAYVIEW`, to_number(pdDistrict = "RICHMOND") AS `RICHMOND`, to
         _number(pdDistrict = "CENTRAL") AS `CENTRAL`, to_number(pdDistrict = "INGLESIDE") AS `INGLESI
         DE`, to_number(pdDistrict = "NORTHERN") AS `NORTHERN`, to_number(category = "VEHICLE THEFT")
         AS `VEHICLE THEFT`, to_number(category = "WARRANTS") AS `WARRANTS`, to_number(category = "WEA
         PON LAWS") AS `WEAPON LAWS`, to_number(category = "DRUG/NARCOTIC") AS `DRUG/NARCOTIC`, to_num
         ber(category = "SEX OFFENSES, FORCIBLE") AS `SEX OFFENSES, FORCIBLE`, to_number(category = "F
         ORGERY/COUNTERFEITING") AS `FORGERY/COUNTERFEITING`, to_number(category = "MISSING PERSON") A
         S `MISSING PERSON`, to_number(category = "VANDALISM") AS `VANDALISM`, to_number(category = "D
         RUNKENNESS") AS `DRUNKENNESS`, to_number(category = "STOLEN PROPERTY") AS `STOLEN PROPERTY`,
         to_number(category = "SUSPICIOUS OCC") AS `SUSPICIOUS OCC`, to_number(category = "DISORDERLY
         CONDUCT") AS `DISORDERLY CONDUCT`, to_number(category = "NON-CRIMINAL") AS `NON-CRIMINAL`, to
         _number(category = "TRESPASS") AS `TRESPASS`, to_number(category = "FRAUD") AS `FRAUD`, to_nu
         mber(category = "LARCENY/THEFT") AS `LARCENY/THEFT`, to_number(category = "OTHER OFFENSES") A
         S `OTHER OFFENSES`, to_number(category = "ROBBERY") AS `ROBBERY`, to_number(category = "ASSAU
         LT") AS `ASSAULT`, to_number(category = "BURGLARY") AS `BURGLARY`, to_number(category = "KIDN
         APPING") AS `KIDNAPPING`, to_number(category = "PROSTITUTION") AS `PROSTITUTION`, to_number(c
         ategory = "SECONDARY CODES") AS `SECONDARY CODES` FROM (SELECT VALUE OBJECT_REMOVE(t, 'id') F
         ROM (SELECT VALUE OBJECT_REMOVE(t, 'location') FROM (SELECT VALUE OBJECT_REMOVE(t, 'incidntNu
         m') FROM (SELECT VALUE OBJECT_REMOVE(t, 'description') FROM (SELECT VALUE OBJECT_REMOVE(t, 'a
         ddress') FROM (SELECT VALUE t FROM to_array(r) AS t) t) t) t) t) t) t) t) t) t) t) t) t)
         t) t) t) t) t)[0]};
```

**Figure 8: Persisting the transformation**

Finally, applying the data transformation and the ML model on a large distributed dataset can be done through AFrame using the apply function. Figure 9 shows the user accessing a different dataset, called 'Crimes', from AsterixDB and applying the 'is_resolvable' function to it. In input line 43, our user filters only the crime incidents that happened in the Central district, possibly assisted under the hood by an index on 'pdDistrict', before applying the trained ML model. The model's prediction results are appended to the dataset as a new column called 'is_resolvable' in line 44. These results can be used to do further analysis or to visualize them using Pandas' compatible visualization libraries by converting the AFrame object into a Pandas DataFrame.



```
In [42]: other_data = AFrame(dataverse='SF_CRIMES', dataset='Crimes',
                             connector=AsterixConnector('localhost:19002'))

In [43]: other_data = other_data[other_data['pdDistrict'] == 'CENTRAL']

In [44]: other_data['is_resolvable'] = other_data.apply('is_resolvable')

In [45]: other_data.head(2)
Out[45]:
```

|   | is_resolvable | incidntNum | category | description | dayOfWeek | date | time | pdDistrict |
|---|---|---|---|---|---|---|---|---|
| 0 | 0.101139 | 186000431 | LARCENY/THEFT | GRAND THEFT FROM LOCKED AUTO | Sunday | 2017-12-31 | 16:00 | CENTRAL |
| 1 | 0.106660 | 186000522 | LARCENY/THEFT | GRAND THEFT FROM LOCKED AUTO | Sunday | 2017-12-31 | 22:35 | CENTRAL |

**Figure 9: Model inferencing**

## 3.4 Lessons Learned

The case study is not only helpful in proving the usability of AFrame but it also helps us identify useful and missing features. For example, the transformation saving mode was created to record the data transformation steps as a function so that they can easily be applied to other datasets using the existing function syntax. Unique Pandas' functions (e.g., describe, get_dummies) are implemented in AFrame by internally calling multiple simple operations in a sequence. This was influenced by the engineered features that the user manually created.

## 4 CONCLUSION

In this short paper, we have detailed an end-to-end case study that utilizes AFrame in a data analysis. We have demonstrated the flexibility and simplicity of using AFrame to perform data preparation, modeling, and deployment on different dataset sizes and moving seamlessly from a small dataset to datasets at scale. We believe that AFrame can deliver a scale-independent data preparation and analysis experience while requiring less effort from data analysts, allowing them to focus on more critical tasks.

Currently, we are extending AFrame by retargeting its incremental query transformation process onto multiple database systems to make its benefits available to existing users of other databases. Our re-architected version of AFrame called PolyFrame, can operate against AsterixDB using SQL++, PostgreSQL using SQL, MongoDB using MongoDB's query language, and Neo4j using Cypher. More information about PolyFrame including its architecture and language configuration rules can be found in [14].

## REFERENCES

[1] Apache AsterixDB. https://asterixdb.apache.org/.
[2] Apache Spark. http://spark.apache.org/.
[3] Dask Anti-Uses. https://docs.dask.org/en/latest/dataframe.html.
[4] Koalas. http://koalas.readthedocs.io.
[5] Pandas. http://pandas.pydata.org/.
[6] Police department incident reports. http://data.sfgov.org/.
[7] Team Data Science Process. https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/overview.
[8] M. Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX OSDI Symposium*, pages 265–283, 2016.
[9] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.
[10] M. Armbrust et al. Scaling Spark in the real world: Performance and usability. *PVLDB*, 8(12):1840–1843, 2015.
[11] F. Pedregosa et al. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12(10):2825–2830, 2011.
[12] M. Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proc. 14th Python in Science Conf.*, pages 130–136, 2015.
[13] P. Sinthong and M. J. Carey. AFrame: Extending DataFrames for large-scale modern data analysis. In *2019 IEEE Big Data Conf.*, pages 359–371, 2019.
[14] P. Sinthong and M. J. Carey. PolyFrame: A Query-based approach to scaling Dataframes (Extended Version). *arXiv preprint arXiv:2010.05529*, 2020.