Time Series Formalism: A Systems Approach

by

Usman Ayyaz

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author ... Signature redacted .....
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by ..... Signature redacted
Devavrat Shah
Professor
Thesis Supervisor

Accepted by ..... Signature redacted
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Time Series Formalism: A Systems Approach

by

Usman Ayyaz

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Time series data has become a modern day phenomena: from stock market data to social media information, modern day data exists as a continuous flow of information indexed by timestamps. Using this data to gather contextual inference and make future predictions is vital to gaining an analytical edge. While there are specialized time series databases and libraries available that optimize for performance and scale, there is an absence of a unifying framework that standardizes interaction with time series data sets. We introduce a python-based time series formalism which provides a SQL style querying interface alongside a rich selection of time series prediction algorithms. Users can forecast data or impute missing entries using a specialized prediction query which employs learning models under the hood. The decoupled architecture of our framework allows it to be easily substituted with any SQL database. We show the functionality of our abstraction with a single machine implementation which will be a building block towards a scalable distributed platform for time series analysis.

Keywords: Time Series, Formalism, SQL, Analytics

Thesis Supervisor: Devavrat Shah
Title: Professor

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Prof. Devavrat Shah for the continuous support of my research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance and belief in me helped me navigate the precarious path of academic research. I am deeply grateful for his mentorship and for the opportunity to have worked with him. Besides my research advisor, I would like to thank my academic advisor Adam Hartz and Professor Helen Lee, who were both a source of guidance and kindness throughout my 5 years at MIT.

I thank my fellow lab mates: Jehangir Amjad, Michael Fleder and Anish Agarwal for the stimulating discussions and their contribution towards our mutual project. Special thanks to my friends Arsalan Adil, Ali Abdalla, Suniyya Amna Waraich and Noor Eddin Amer for their friendship,camaraderie and continued support which made this journey possible. To Hajar Boughoula, for her friendship and unwavering belief in me. To Nadeen Abuhasan, for being a trusted friend and a source of optimism and strength. To Atif Javed for honest advice and Abubakar Abid for setting an example of excellence. To Hoda Elsharkawi, for spiritual guidance and her friendship. To my friends, who I left behind in Pakistan for academic pursuit - I miss you every day. To my school teachers, Dianne Grady McKenzie and Sabahat Zakariya, for instilling in me a passion for excellence through their dedication and professionalism.

Last but not the least, I would like to thank my family: my mother Rooh Afza, for her lifelong sacrifices and prayers, my father Muhammad Naeem Ayyaz, for being a trusted guide, for instilling in me a passion for excellence, for teaching me how to dream and for your lifelong hard work which has made this moment possible; I will forever be indebted to your sacrifices. To my nephew, Affan for being a bundle of joy and to my brother Ali for being an inspiration of resilience and fortitude. To my sister Mushal Noor for always pushing me on and believing in me, for sharing her strength with me and for the most trusted of advice. And to Iqra Ali and Saad Javed for the blessing of family.

Contents

Chapter 1

Introduction

Predicting customer intent on the web is an area of growing interest with significant impact on revenue and customer satisfaction. In today's highly competitive economy, understanding online customer intent accurately and at an infinitesimally short timescale is a significant competitive advantage. Predicting intent for users interested in exploring, purchasing and troubleshooting, for example, has a direct impact on providing the best automated service to the customer. This understanding and inference of context and predictions for the future can be referred to as Time Series Analytics.

Time series analytics broadly encompasses two different categories of problems: imputing missing data and forecasting future data. However, data sets often have missing entries due to system failure, network outage or database issues, which result in irrecoverable data loss. Filling up these data holes in the past using imputation methods brings us one step closer to a complete data set. However, the real value of time series algorithms lies in predicting the future. Trends that appear in the past continue through in the future as well and the ability to capture these trends accurately and precisely is at the core of time series predictions. The former is necessary to gain a complete picture of the past and the latter is at the core of many financial applications that provide businesses a competitive edge.

Time series data is of enormous interest across all domains of life: from health sciences to weather forecasts to retail and econometrics, time dependent data are

everywhere. However, time series analysis and forecasting are hard problems, albeit well-studied. The primary departure from standard methodologies for statistical inference is due to the dependencies across time which renders much work tailored for independent (and often identically) distributed data to be less relevant without mitigating conditions. This dependence across time is modeled in several ways: periodic time series display a cyclical or "seasonal" pattern; trends display a growth or decay with time or more generally long term relationship in data; and stationary auto-regressive models capture dependence on the very recent past. For each type of time series, there exists a rich body of work which allows a practitioner to model observed data and predict in to the future.

In practice, the prominent approach is to utilize a linear combination of the periodic, trend and auto-regressive models to model the time series. Such an approach requires iteratively "peeling off" trend, periodic and auto-regressive model structure using data to produce an eventual combined model. For each stage of peeling off, this requires evaluating some form of 'model score' across model types and parameters to choose the best fit. While feasible in principle (and used in practice), by design it is a hodgepodge that makes it painstaking for the practitioner, as well as computationally and statistically inefficient. Further, the state-of-art has limited guidance in terms of how to combat the issue of missing data – natural options being either discarding some of the data or "filling" missing data by default values or empirical means.

Applications around time series data also introduces a problem of scale, since the data is often sampled on the scale of milliseconds or nanoseconds in the case of stock market financial data sets. The total storage required to store a year worth of data for a single publicly traded stock borders $\sim$ 100 TB. Data at such massive scale introduces unique challenges that directly impacts the latency of inference which is uncompromising in many applications, e.g. high frequency trading. Moreover, the accuracy of learning and predictions at such a scale becomes a challenge because classical algorithms need to be rethought from a context time-aware perspective.

In the online world today, time-series context resides in a very wide array of disparate sources and there is a need for a unifying formalism that provides a theoret-

ically powerful and practically implementable framework. In this paper, we propose a time series formalism that unifies the prediction problem and describe the systems framework surrounding it. Our aim is to introduce an all encompassing formalism that simplifies the routines implemented on time series data sets and move one step closer to a complete end to end solution.

Chapter 2

Literature Review

Time series analysis is an integral part of econometrics and the classical auto-correlation models, both parametric and non-parametric, used for forecasting and smoothing have been thoroughly studied [1]. Departing from the conventional box-jenkins modeling, alternative approaches using hybrids of artificial neural networkds [2] and support vector machines[3] have also been successfully employed for time series forecasting. However, more recently with the onset of big data and the computational complexities that come along with it, a systems approach towards time series analysis has become incumbent. The widespread usage of query optimized time series databases like KDB+ q [4] in the financial business and popularity of specialized time series packages point towards a real need for tools that can address industry requirements. In the following sections, we provide a brief overview of existing solutions in time series analysis.

2.1 Methods for Time Series Analysis

In statistics and signal processing, a time series is a sequence of data points indexed at strictly increasing observation time steps in a regular orderly fashion. Conversely, in irregular time series, the spacing of observation times is not evenly spaced. A common approach to analyzing unevenly spaced time series is to transform the data into equally spaced observations using interpolation methods [5], e.g. estimating missing values by decomposing the problem as a combined prediction and regression problem [6],

and then treating it as any other regular time series . However, transforming data in such a way can potentially introduce biases especially if the spacing of observations is highly irregular.

Broadly speaking, the methods for time series analysis can be divided into two distinct classes: frequency-domain methods and time-domain methods. The former include spectral and wavelet analyses [7]; the latter include auto-correlation and cross-correlation analyses. In the time domain, correlation and analysis can be made in a filter-like manner using scaled correlation, thereby bypassing the need to operate in the frequency domain. Additionally, time series analysis techniques may be divided into parametric and non-parametric methods. The parametric approach assumes that the underlying stationary process has a certain structure which can be modeled by parameters, e.g. the auto-regressive and moving average parameters in ARMA/ARIMA models. In this approach, the task is to learn the model parameters that describes the stochastic process. By contrast, non-parametric approaches explicitly estimate the covariance of the process without making any structural assumptions about the process [8]. For the rest of this paper we primarily focus on time-domain methods and both univariate and multivariate data sets.

2.2 Time Series Libraries

As previously discussed, there are very few industry standard open source time series algorithm packages available with the few exceptions limited to Python. Libraries like Numpy and Pandas provide some basic functionality for manipulating time series objects. Statsmodels [9] builds on top of scipy and provides a good collection of models ranging from uni-variate auto-regressive models (AR), vector auto-regressive models (VAR) and uni-variate auto-regressive moving average models (ARMA) [10]. Pyflux, also built on top of scipy, shares similar functionality to statsmodels with some added models like generalized autoregressive conditional heteroscedasticity (GARCH) and generalized autoregressive score (GAS) models [11].

Outside python, R provides an extensive time series analysis library with methods

ranging from uni-variate/multi-variate modeling to frequency analysis to decomposition and filtering of time series, as well as non linear models [12]. Despite boasting a rich library, R is not as flexible as Python in providing programming interfaces for application development. Moreover, the object oriented programming paradigms offered by Python lend more abstraction power than R, making it an ideal candidate for developing our formalism. Furthermore, we envision systems framework to eventually bridge with Spark for scalable distributed computation and the existing PySpark [13] interface between Python and Spark will help in future development of the project.

2.3 Time Series Databases

Time Series databases (TSDB) are optimized for handling storage and retrieval of time series data. TSDB take advantage of the orderly and increasing nature of time series data sets to minimize storage overhead and design performance optimized queries. Scalability and performance are key design metrics for TSDB as applications need to scale easily to support data in a continuous flow and perform real-time analysis. Queries for historical data, with complex time ranges, roll ups and time zone conversions are difficult in a relational database. TSDBs on the other hand impose a model and this allows them to provide more features for doing so.

InfluxDB [14] is an example of a specialized time series database developed by InfluxData. It optimizes for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics. OpenTSDB [15] is another scalable time series database built on top of Hadoop and HBase with plugins for Spark as well. It simplifies the process of storing and analyzing large amounts of time-series data generated by endpoints like sensors or servers. Both these databases provide examples of current TSDB solutions in the industry which focus on optimizes storage and retrieval of time series data. For our system, we use a non-TSDB SQL based database to show the applicability of our abstraction. However, in the future we will explore TSDB for performance optimization in algorithms.

Chapter 3

Technical Work

Definition 1. A time series model for the observed data x_t is a specifi- cation of the joint distributions (or possibly only the means and covariances) of a sequence of random variables X_t of which x_t is postulated to be a realization. [16]

From a classical perspective, a time series X_t can be broken down into four constituent parts as follows:

- Trend T_t : long term movement in the mean e.g. linear or polynomial trend
- Periodicity P_t : cyclical fluctuations that repeat regularly over time
- Stationarity S_t : stochastic process with fixed probability distribution when shifted in time.
- Residuals R_t : random noise and systematic fluctuations

These constituent parts combine to form a class of time series which could be summarized under our formalism. Our goal is to create a systems framework that provides the necessary tools to manipulate a broad class of time series instances. In the next section we define the model class of time series in more detail.

3.1 Time Series Formalism

We shall consider discrete-time time series data: let $X_t \in \mathbb{R}$ denote the value of the time series at time $t \in \mathbb{Z}$. A prevalent approach in time series literature for modeling is to decompose its dynamics into the following components: (1) stationary-auto-regressive; (2) periodic; (3) trend; (4) per-step independent noise. Therefore, we consider the following class of time-series: over an interval $[T] = \{1, \ldots, T\}$ (with notation $X[s\colon t] = (X_s, \ldots, X_t)$)

$$F_T = \Big\{ X[1\colon T] : X_t |\ X[1\colon t-1] \stackrel{d}{=} \boldsymbol{h}(t; \alpha, p) + \boldsymbol{g}(t; \gamma) + \boldsymbol{\varepsilon}(t),\ \forall t \Big\}, \tag{3.1}$$

where $\boldsymbol{h}(t; \alpha, p) = \sum_{i=1}^{p} \alpha_i X_{t-i}$ is a deterministic linear function of $X[t-1\colon t-p]$ with $p \geq 1$ parameters $\alpha = (\alpha_1, \ldots, \alpha_p)$ such that $\sum_{i=1}^{p} |\alpha_i| < 1$; $\boldsymbol{g}(t; \gamma)$ is any deterministic Lipschitz function of time t; and $\boldsymbol{\varepsilon}(t)$ are independent and identically distributed random variables with mean 0 and variance $\sigma^2 < \infty$ representing noise terms.

Given the conditional distribution $X_t | X[1 : t-1]$ per $\boldsymbol{\mathcal{F}}_T$, we have that

$$\mathbb{E}[X_t | X[1\colon t-1]] = \boldsymbol{h}(t; \alpha, p) + \boldsymbol{g}(t). \tag{3.2}$$

Next we argue that several popular time-series models belong to this model class.

Auto-Regressive (AR). The standard auto-regressive (AR) model with parameter $p \geq 1$ and co-efficients $\alpha = (\alpha_1, \ldots, \alpha_p)$ obeys

$$X_t = \sum_{i=1}^{p} X_{t-i}\alpha_i + \boldsymbol{\varepsilon}(t) \ = \ \boldsymbol{h}(t; \alpha, p) + \boldsymbol{\varepsilon}(t). \tag{3.3}$$

That is, by setting $g(t, \gamma) = 0$, any AR(p) belongs to the model class introduced. It is worth remarking that for well-behaved AR(p) model, $\sum_{i=1}^{p} |\alpha_i| < 1$ holds.

Periodic. A periodic or mixture of finite frequency model obeys

$$X_t = \sum_{k=1}^{K} \eta_k \sin(2\pi\omega_k t) + \nu_k \cos(2\pi\omega_k t) + \boldsymbol{\varepsilon}(t). \tag{3.4}$$

That is, by setting $\boldsymbol{h}(t;\alpha,p) = 0$ and with $\gamma = ((\eta_k, \nu_k, \omega_k))_{1\leq k\leq K}$,

$$\boldsymbol{g}(t;\gamma) = \sum_{k=1}^{K} \eta_k \sin(2\pi\omega_k t) + \nu_k \cos(2\pi\omega_k t), \tag{3.5}$$

we have that it belongs the model class since $\boldsymbol{g}$ can be verified to be a Lipschitz function.

Trend. This class of time series obeys

$$X_t = \sum_{k=0}^{K} \gamma_k t^k + \varepsilon(t). \tag{3.6}$$

That is, by setting $\boldsymbol{h}(t;\alpha,p) = 0$, with $\gamma = (\gamma_k)_{1\leq k\leq K}$ and $\boldsymbol{g}(t;\gamma) = \sum_k \gamma_k t^k$, we have that it belongs to the model class defined above since $\boldsymbol{g}$ can also be verified to be a Lipschitz function. The model class F_T described here sets up the backbone of our formalism and defines the different constituent parts that describe any time series instance. This decomposition allows us to create an encompassing algorithmic interface that can interact will all the moving parts of a time series and employ learning models. In the next section we describe the system architecture that implements the mentioned algorithmic interface.

3.2 System Design

From data logging to storing sensor output to tracking economic trends, a diverse set of applications naturally fall under the umbrella of time series model. Businesses employ applications that analyze time series of user metrics to extract behaviour patterns that can provide them insights into their products. Extracting these insights, however, requires a robust framework with performance optimized database queries, efficient ETL pipelines, context aware algorithms and intuitive visualizations. We propose a basic system design in 3-1, where time series data is collected in a database, PostgreSQL in our case, and a learning model is trained on a sub set of this data to create predictions in the future.

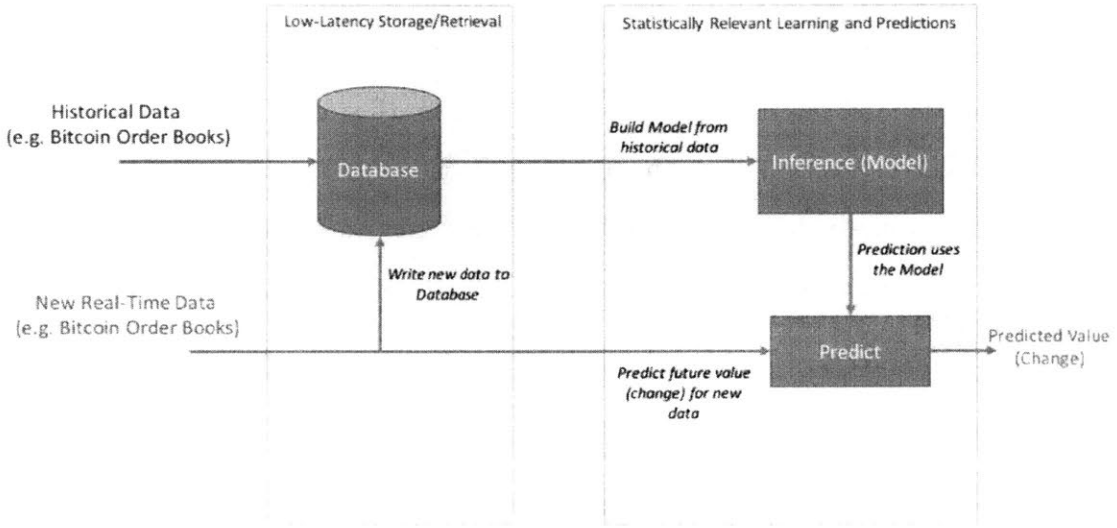

Figure 3-1: Framework

Establishing an interface that can store time series instances of F_T, retrieve data and train models is necessary to implement the formalism. Our proposed interface aims to formalize interaction with the database by defining a class of queries that integrate learning models with the data. We use SQlAlchemy [17], an object relational mapper, to define an interface which can be used with any SQL database. SQLAlchemy provides a rich set of enterprise level persistence patterns to access SQL databases

through Pythonic domain language. The interface in itself has the ability to pipeline with any computing stack. This requires a modularity in approach which is at the core of our system design. The diagram below summarizes the main components of our interface and their interactions.

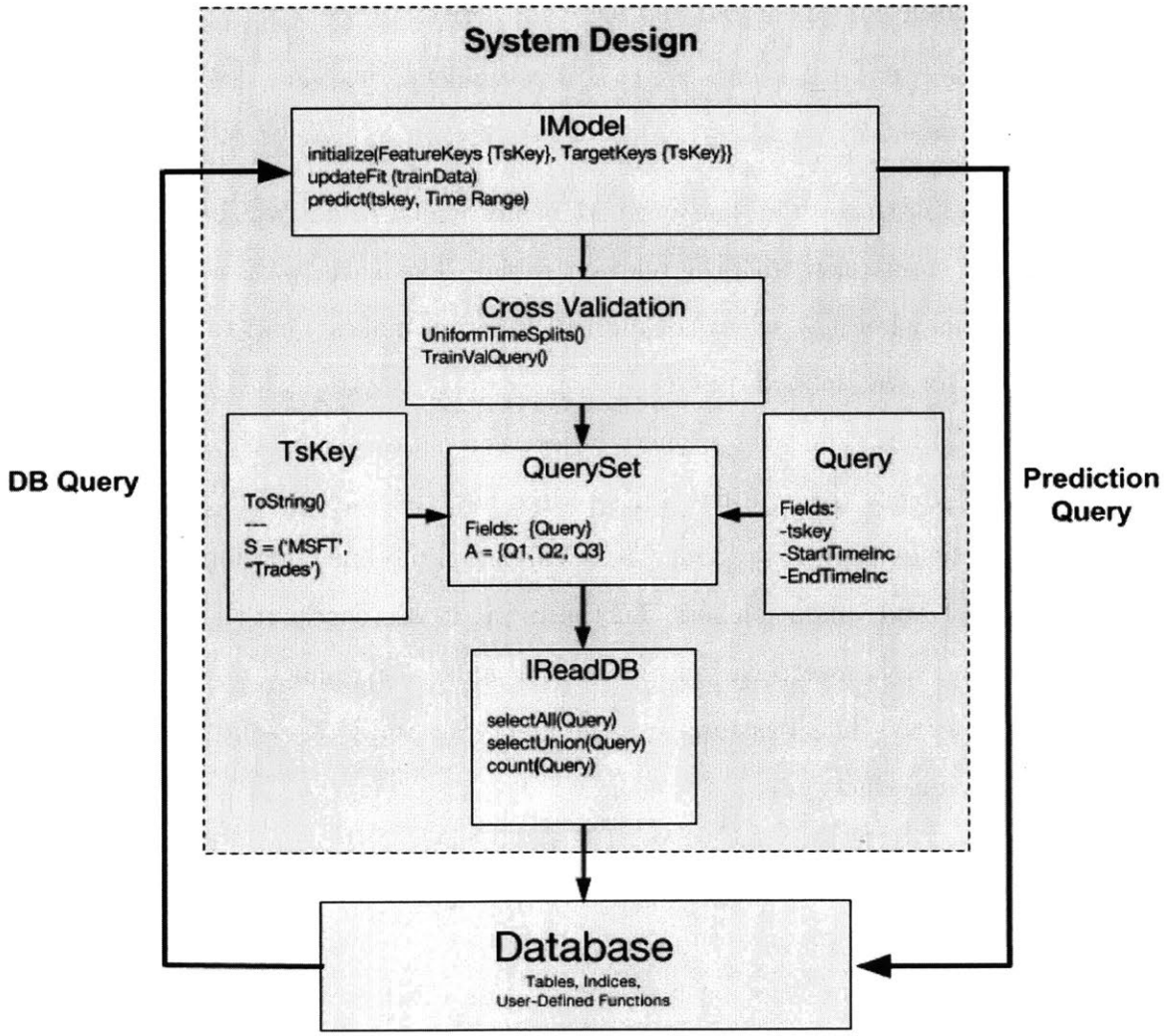

Figure 3-2: System Design

As shown in 3-2, our system consists of independent modules with minimal dependence. The abstractions highlighted in the grey box show the different modules that work together to communicate with the database. The Model represents the different learning algorithms in our library e.g., ARMA, USVT with Matrix Completion,

Random Forest etc. The algorithm takes in a train query partitioned by the Cross Validation module and learns the model. To initialize the model, we specify the input time series using the TsKey identifier, and the target time series which will be predicted by the model. For example, we initialize the ARMA model to predict APPL stock price as a function of the bid and ask time series:

```
ARMA.initialize({Feature=APPL_{bid}, APPL_{ask}}, Target= {APPL_{price}})
```

The QuerySet consists of a set of queries defined by the model that are upheld by the ReadDB interface. The query set identifies which tables will be queried and sets up the data connection with the relevant tables. The actual data access from the database happens when the selectAll(queries) function makes a call to the database to retrieve data for the defined queries.

The selectAll() function takes a Query object as a parameter which is identifies the relevant table to query through the TsKey identifier, the start and end times of the query and what fields to retrieve. Our Cross Validation module partitions the data set into training, test and validation sets. Furthermore, it also checks if the queried data is regularized and uses interpolation methods to make it regular if not. A describes how to connect to the data base and use the QuerySet and CrossValidation modules explained in our interface.

```
from datetime import datetime
from queries import *
from pg_tskey import TsKey
import numpy as np
from uniform_time_splits import UniformTimeSplits
from postgres import Postgres

### Example 2: Setting up a generic Postgres query ###

data_source = Postgres(dataset="synthetic")
tskey = TsKey(name="arma_process", fields=['timestamp'])
start_date = datetime(2017, 3, 19, 21, 40, 3, 515734)
end_date = datetime(2017, 3, 19, 22, 40, 2, 515734)
datasetQ = Query(tskeys={tskey}, start_time_inc=start_date, end_time_inc
```

```
    =end_date)
time_delta_bw_pts = np.timedelta64(microsecond=1)
train_val_split = UniformTimeSplits(QuerySet({datasetQ}), trainPerc=0.8,
    min_time_delta_bw_pts=time_delta_bw_pts / 2)
#training and validation set
train_df_orig = data_source.select_all_union(train_val_split.getTrain())
val_df_orig = data_source.select_all_union(train_val_split.getVal())
```

The Model class encompasses a growing set of time series algorithms that are used for forecasting timeseries. Here, we briefly cite a USVT based matrix completion forecasting algorithm that was designed with our interface

Time Series Through Matrix Completion

We model a broad class of time series data F_T as a matrix and then show that it has an approximate low-rank structure. This structure allows the use of a popular matrix completion technique, specifically singular value thresholding, to "de-noise" the observation matrix to obtain a low-rank estimation. We then use linear regression to determine the appropriate relationships between the rows of the estimated matrix. This relationship is then used to forecast future values of the time series. Our algorithm provides a principled, robust way to address the challenge of missing data for the broad model class considered here. The algorithm does not need tuning to adapt to different types of model parameters class under consideration. We provide finite sample analysis to guarantee good approximations in terms of the average mean squared error. Experiments confirm excellent performance, especially in settings with missing data. This paper has been submitted to NIPS for submission [18].

Our library also includes classical ARMA/ARIMA modeling and other basic time series algorithms which were inherited from statsmodels and pyflux libraries and wrapped with our interface. Together these algorithms form the basis of a comprehensive time series library which we continue to expand in the future.

3.2.1 Query Interface

As part of the system design, training data is retrieved from the database for the learning model and the subsequent predictions are cached back into the database for the user to query. As new data comes in the model is retrained at fixed intervals and the model state is stored in the system to make predictions. This helps us provide real time predictions in a latency sensitive manner. In order to achieve modularity, we introduce two different notions of querying in our interface - DB Query and Prediction Query.

Applying learning models on F_T requires a robust framework that standardizes the time series storage, retrieval, transformation and prediction operations through a simple interface. We take a SQL style querying approach towards designing this interface, where a user interacts with a time series $X[1:n]$ as if it is an existing table in the database. A query for $t > n$ is predicted by a learning model specified by the user and cached in the database as an update time series $X[1:t]$. The query interface consists of a DB query and a Prediction query, the former returns existing data from the database and the latter predicts data that is not available in the time range

DB Query.

Our interface expects unit data to be stored as a (type, time range) tuple with an associated value which can be d-dimensional. Hence, we can define the unit query as follows

Argument: (type: s; timestamp: $[a:b]$)

Response: $X[a:b]^d$ i.e. d-dimensional values in range $[a:b]$

For example we can query stored values of "AAPL" stock in the time range $[a:b]$ as follows:

```
SELECT * FROM "APPL"
WHERE [timestamp] > 'start_time' AND [timestamp] <= 'end_time';
```

Prediction Query.

Data that is not stored in a table can be predicted by a user defined model and cached

in the database.

Argument: (type: s; time range:[a:n])

Response: $G(X_t[a:b])) \rightarrow X_t[a:b], Y_t[b+1:n]$

Here, we define the prediction function G to be any user specified predictive model, e.g. Random Forest, Linear Regression etc which takes $X_t[a:b] \in F_T$ as an input to the training model and predicts $Y_t[b+1, n]$ in the future as a function of previous values X_b, X_{b-1} and so on. A sample prediction query on an APPL stock utilizing a random forest prediction model would be as follows:

```
SELECT * FROM "APPL"
WHERE [timestamp] > 'start_time' AND [timestamp] <= 'end_time' AND
[model] = "random_forest"
```

3.3 Database

To provide a complete end to end solution, it is necessary to test your interface with data sets of varied scale. Our goal is to build a system that has the capacity to process data on the scale of TB. This would be a novel contribution, since none of the existing time series libraries like scikit-learn and statsmodels etc, provide latency sensitive processing at such scale. A significant effort was made on our behalf to research and host data sets that are relevant to time series analysis. In this section we explain the data sets in detail and also introduce the computation stack and database used.

3.3.1 Azure

As mentioned before, we envision the final version of this system to be a distributed scalable platform for time series analysis. Hence, an obvious design choice was to leverage cloud computing platforms. Azure was a stand out choice as it offers enterprise level tools for shared computation and database hosting. Moreover, the customized configurations for SQL style databases and seamless integration with Apache Spark tied up perfectly with the anticipated future work.

We decided for a performance optimized single machine implementation of our interface to act as a benchmark to future implementations of a distributed architecture. We used an enterprise 4 Core machine with 56 GB RAM, Linux operatingj system. In addition, we attached 3 SSD hard drives of 1 TB each to store our data sets and configured out stack for a network optimized 518200 max IOPS.

3.3.2 PostgreSQL

The design decision to use PostgreSQL as our database primarily stemmed from a need to have SQL styled query interface for time series data sets. Since, time series have a fixed schema with a column for timestamps and other associated fields, it immediately eliminated NoSQL databases from the competition. Moreover, PostgreSQL's MVCC data storage model is optimized to for large volumes which was of direct interest to us given the TB scale of data. Unlike, other databases like Cassandra, PostgreSQL does not put a limit on the number of tables in a database, neither does the performance decline with increasing number of columns. Both of these consideration were important design decision as will become apparent as we discuss the data schema. Finally, PostgreSQL provides a timestamp field datatype which can store time fields at nano-second granularity. This was a necessary feature for us, since financial data sets like stock index are sampled at nano-second intervals.

As discussed in the introduction, there are many time series optimized databases in the market now, however, PostgreSQL have certain definitive advantages:

- Longevity: SQL has been around for almost 50 years and the trend is expected to continue in the future.

- Ubiquity: It is the most common SQL server with most applications in the industry.

- Community: It has a well established community which will be essential in building our open source library.

Due to its universality, PostgreSQL provided a solid testing ground to implement our formalism and easily make it accessible to the community. Since our query interface is developed using an object relational mapper, we have the flexibility to integrate our abstraction with any SQL based database. As more concrete use cases our library are developed, we will explore further database options including possible TSDB solutions.

3.3.3 Youtube8M

YouTube-8M is a large-scale labeled video dataset that consists of millions of YouTube video IDs and associated labels from a diverse vocabulary of 4700+ visual entities. It comes with precomputed state-of-the-art audio-visual features from billions of frames and audio segments, designed to fit on a single hard disk. This makes it possible to get started on this data set by training a baseline video model in less than a day on a single machine! At the same time, the data set's scale and diversity can enable deep exploration of complex audio-visual models that can take weeks to train even in a distributed fashion. [19]

Here are a few statistics about the data set:

- Each video is public and has at least 1000 views
- Each video is between 120 and 500 seconds long
- Each video is associated with at least one entity from our target vocabulary

The target vocabulary consists of labels for each video. A video can have multiple labels associated with e.g. {'cat', 'dog', 'tree'}. The YouTube data set is divided into two genre, video level aggregated features and frame level features. For both genre, we have the rgb and audio features of the video available. However, for aggregated video features, we have the mean-rgb and mean-audio for the whole video averaged over all the frames. Conversely, in the frame level features, we have the mean and audio features for each frame available. It is to be noted that video are of variable length ranging anywhere from 2 to 5 minutes. The features are generated from a

convolutional neural network and then PCA-ed for compression to return 1024 8-bit quantized features for each frame. [19]

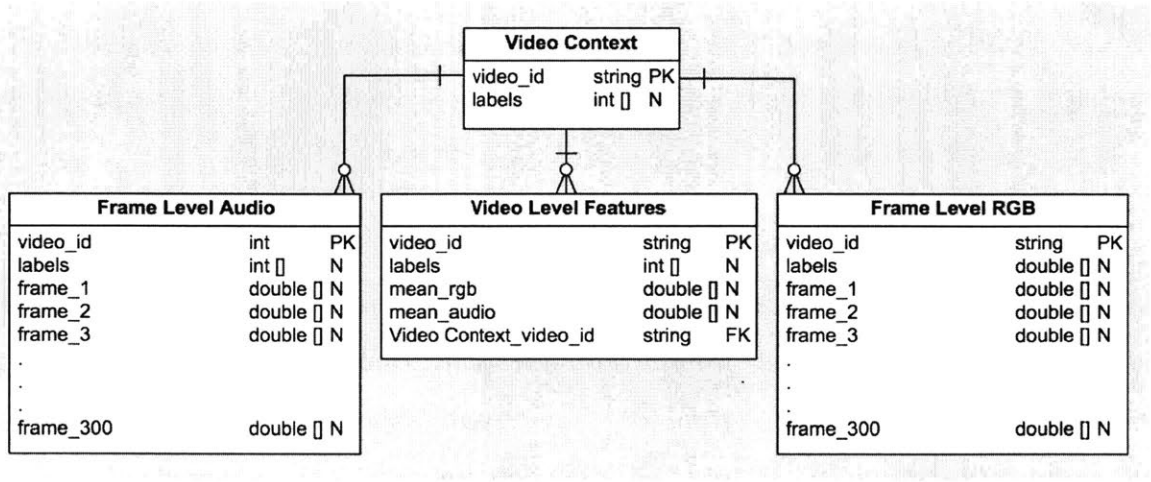

Figure 3-3: YouTube Video Dataset Schema

The YouTube8M data set is already partitioned into test, train and validate sets. For the test set the labels are not provided. For each of these partitions, we have the data schema shown in 3-4. Each entry is identified by a primary key which is the video ID. The video ID is part of the Video Context table which is referenced by all the other tables. We made a design decision to store RGB and Audio features in separate tables to decrease the width of each table respectively. Currently each frame column stores all the 128 8-bit quantized features for that frame, so decreasing the table width in half makes page retrieval in PostgreSQL faster.

Tensorflow to PostgreSQL

The YouTube8M data set is publicly available as tfRecord files to be used in TensorFlow. In order to access the data, we first transformed the tfRecord file to a txt file and extracted the quantized features after string manipulation. The transformation scripts along with schema files were made publicly available to users who want to store the YouTube data set in a SQL data base.[20]

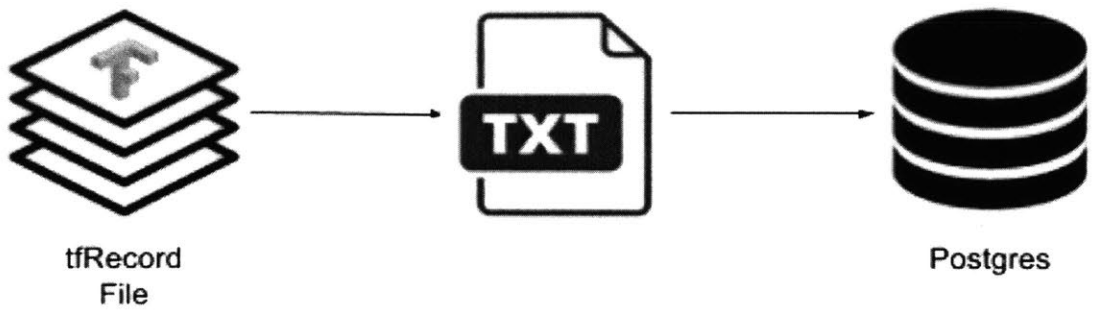

Figure 3-4: tfRecord to Postgres Data Plug

Synthetic Data

We also provide access to synthetic data to test system performance for particular conditions which might not be found in the real data. Moreover, development of new algorithms require sanity checking which can be done by evaluating performance on self-defined synthetically generated datasets. For example, we provide sample ARMA data generators that were used in the testing and development of the USVT method described above. We also provide generic data insertion scripts in SQLAlchemy that let us store any data set with minor changes. This gives us the flexibility to generate appropriate data for our algorithms and assess their performance in a closed environment.

3.4 Evaluation

As our system evolves, we will be continuously evaluating it against predefined metrics to ensure we reach our design goals. In light of our initial design motivation, we will be testing our system against the following metrics:

- Scale: One of the main motivations behind our system is to create a distributed platform that can process data at a high scale. Current open source libraries struggle to ingest data beyond multiple GBs. In comparison, we aim to process data on the scale of TBs in an efficient latency sensitive manner.

- Performance: Another goal of this system is to implement distributed version of classical algorithms specific to time series. Using the vanilla implementations of these algorithms as a benchmark, we will test the performance of our distributed implementations to ensure that our accuracy is up to mark.

- Robustness Our system design is required to be fault tolerant with redundancies in place for system failure. We will be testing our system for robustness and its ability to take in data sets of varied scale and pipeline them with our suite of algorithms.

Implementing the USVT algorithm for matrix completion using our interface provided us with benchmarks to evaluate our system. Here we describe the results of those experiments to validate that our interface has the capacity to act as an end to end solution for time series analysis.

Predicting Temperatures in Melbourne

In order to study the applicability of our algorithm to real-world time series data, we consider predicting the daily low temperatures in Melbourne, Australia in the year 1990 using the 1981-1988 period to train and 1989 to validate. Note that while the synthetic results were in the large data regimes, this dataset is relatively tiny with about 3600 data points. As with all real-world datasets, we are unaware of the *true* data generating model or levels of noise. We convert the training data points in to a matrix of dimensions 40×3200 and reserve the remaining 400 data points for testing. Figure 3-5 shows the entire data set (1981-1990) in gray and our in-sample and out-of-sample predictions in green. Notice that our algorithm produces excellent qualitative performance.

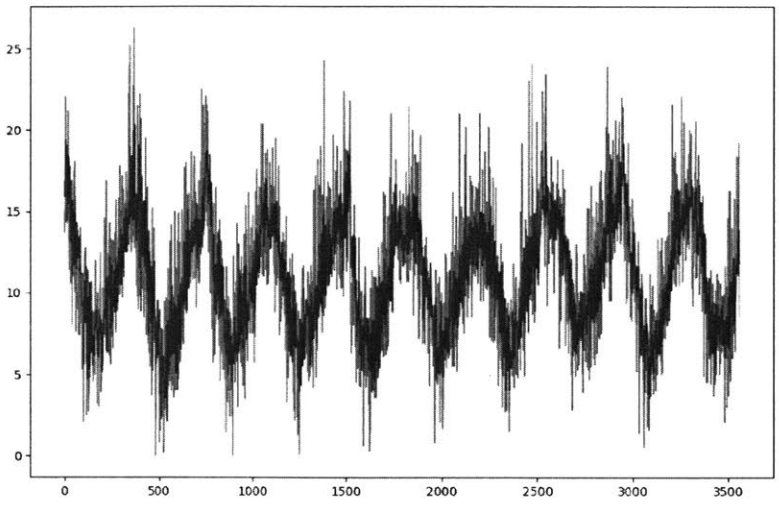

Figure 3-5: Minimum daily temperatures in Melbourne, Australia from 1981-1990. The two out-of-sample prediction profiles are shown in green.

The above results show that our system is robust enough to integrate a new algorithm and get good results. As our system moves towards distributed architecture for scalable computation, the scalability metric will come into focus.

Chapter 4

Future Work

The time series formalism introduced in this thesis sets up a framework for time series learning and prediction algorithms. Our aim is to build a system that utilizes this framework for scalable learning and prediction of large scale time series data sets. This requires both novel algorithms that leverage the fundamental properties of time indexed data for learning purposes and a robust distributed system that has the processing prowess to provide near real-time predictions on TBs of data. The system design proposed in this paper attempts to set benchmarks through a single machine implementation. Once we have a proof of concept for a single machine, we can start designing the architecture for a multi-machine distributed platform. The future steps in this project can be summarized as follows:

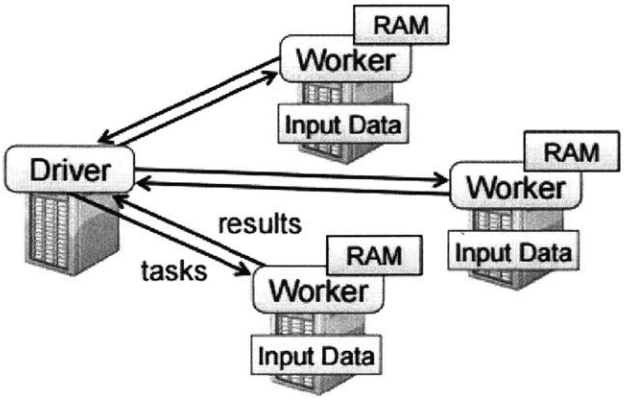

Figure 4-1: Spark architecture for distributed computation

- Spark: Implement the time series formalism in a distributed architecture using Apache Spark or similar platforms as shown in 4-1.

- Database: Optimize query performance by caching. Provide custom database functions which users can use to transform data at the storage source and minimize I/O transfer volume. Research alternative data store methods like HDFS file system or Azure data lake.

- Algorithms: Implement distributed versions of existing classical algorithms and provide new algorithms for time series forecasting. Extend the current formalism and add more functionalities to the library including, an extended cross validation module, lag and shift operators for time series manipulation and generic helper methods like L1/L2 norm and other loss functions for algorithm development.

- Metrics: Define a new database performance metric that compares the ratio between the retrieval time for DB and Prediction queries. This metric will become a standard for all algorithms implemented in the library

Chapter 5

Conclusion

The widespread prevalence of time series data has created a need for a generic and scalable analysis platform that specializes in time series specific algorithms. Our aim was to design an open source platform that adheres to a fixed interface which formalizes large scale processing and retrieval of time series data and works as a building block towards more complex operations and algorithmic approaches to extract information from large data sets. We define a time series model class that adheres to a system abstraction and provides the framework to implement time series prediction algorithms. We show the robustness of our system by showing seamless integration with a USVT based matrix completion method for time series prediction. Furthermore, we extend the data capacity of the system by integrating a 1.7 TB YouTube video data set which will be used to test and build scalable algorithms. We envision our platform to become a building block towards a distributed platform for time series analysis.

Appendix A

Documentation

Postgres Class

```
1  from sqlalchemy.orm import sessionmaker
2  from sqlalchemy.ext.declarative import declarative_base
3  import traceback
4  import sys
5  from iread_db import IReadDb
6  from codeutils import *
7  from datetime import datetime
8  from queries import *
9  from pg_tskey import TsKey
10 import numpy as np
11 from uniform_time_splits import UniformTimeSplits
12 from exceptions import *
13
14
15 class Postgres(IReadDb):
16
17     def __init__(self, dataset):
18         # Provide url to connect to db when query is executed
19         if dataset == "synthetic":
20             self.url = *****
21         elif dataset == "youtube-video-level":
22             self.url = *****
```

```
23          elif dataset == "youtube-frame-level":
24              self.url = *****
25          else:
26              raise DatasetError("Dataset does not exist")
27
28          self.Base = declarative_base()
29          self.db = create_engine(self.url)
30          self.metadata = MetaData(self.db)
31          Session = sessionmaker(bind=self.db)
32          self.session = Session()
33
34      def select_all(self, query):
35          print(type(query))
36          return query.execute(self.session, self.metadata)
37
38      def select_all_union(self, qset):
39          if len(qset.queries) != 1:
40              raise NotImplementedError("not yet handling multiple query
    unions")
41          return self.select_all(first(qset.queries))
42
43      def count(self, query):
44              pass
45
46      def estimate_size_mb(self, query):
47          pass
48
49      def add_timeseries(self, key, start_time, end_time, fields):
50          class Dataset(self.Base):
51              __table__ = Table('datasets', self.metadata, autoload=True)
52          try:
53              record = Dataset(**{
54                  'key': key,
55                  'start_time': start_time,
56                  'end_time': end_time,
57                  'fields': fields
```

```
58              })
59              self.session.add(record)
60              self.session.commit()
61
62          except:
63              print("Exception in user code:")
64              print('-' * 60)
65              traceback.print_exc(file=sys.stdout)
66              print('-' * 60)
67              self.session.rollback()  # Rollback the changes on error
68          finally:
69              self.session.close()  # Close the connection
70
71      def delete_timeseries(self):
72          pass
73
74      def __str__(self):
75          pass
```

Generic Insertion Script

```
1
2 from time import time
3 from sqlalchemy.ext.declarative import declarative_base
4 from sqlalchemy.orm import sessionmaker
5 import csv
6 import numpy as np
7 from tskey import TsKey
8 from queries import *
9 from usvt_arma_generator import USVTArmaGenerator
10
11
12
13 ######## Example 1: Inserting data into POSTGRES ########
14
15 #Initialize connection to to remote database.
16 #Make sure Azure VM is running.
17
18 Base = declarative_base()
19 engine = create_engine(<machine ip address>)
20 metadata = MetaData(engine)
21 #MetaData object where newly defined Table objects are collected
22 engine.echo = False
23
24 """
25 Define Table Structure for the csv import. If you are inserting new
      values to an existing table then simply use __tablename__ = <
      existing table name>. SQLAlchemy provides wrapper classes around
      generic Postgres datatypes e.g. DateTime, Float etc. Datatype
      details can be found here:
26 http://docs.sqlalchemy.org/en/latest/core/type_basics.html#generic-types
27 """
28 class Arma_Process(Base):
29     __tablename__ = 'arma'
30     # tell SQLAlchemy the name of columns and their attributes:
```

```
31      timestamp = Column(DateTime, primary_key=True, nullable=False)
32      value = Column(Float)
33
34  def readCSV(filename):
35      data = []
36      with open(filename, 'r') as csvfile:
37          header = csvfile.readline()
38          reader = csv.reader(csvfile)
39          for i in reader:
40              data.append(i)
41      return data
42
43  if __name__ == '__main__':
44
45      start_time_inc = np.datetime64('2016-12-14 20:30:00.123456789')
46      end_time_inc = start_time_inc + np.timedelta64(3600000, 's')
47      time_delta_bw_pts = np.timedelta64(10, 's')
48      time_series_key = TsKey('taco')
49      dsetQuery = QuerySet.single(
50          Query(start_time_inc=start_time_inc, end_time_inc=end_time_inc,
      tskeys={time_series_key}))
51
52      data_source = USVTArmaGenerator(const=10.0, phi=[-0.4, 0.3, 0.2],
      theta=[], # phi=[-0.2, 0.4, -0.1], theta=[],
53                                      sigma=1.0, burnin_period=np.
      timedelta64(int(1e2), 's'),
54                                      data_spacing_delta=time_delta_bw_pts
      ,
55                                      dset_queries=dsetQuery)
56
57      try:
58          data = data_source.select_all_union(dsetQuery)
59          a = data["taco"].iteritems()
60          for i in a:
61              record = Arma_Process(**{
62                  'timestamp': i[0],
```

```
63                 'value': i[1]
64             })
65             s.add(record)  # Add all the records
66
67         s.commit()  # Attempt to commit all the records
68     except:
69         print("there was an error")
70         s.rollback()  # Rollback the changes on error
71     finally:
72         s.close()  # Close the connection
73     print("Time elapsed: " + str(time() - t) + " s.")
```

tfRecord to SQL Data Plug

```
1
2 from time import time
3 from sqlalchemy.ext.declarative import declarative_base
4 from sqlalchemy.orm import sessionmaker
5 from sqlalchemy.types import String, Integer, Float
6 from sqlalchemy.dialects import postgresql
7 from sqlalchemy.orm import sessionmaker
8 from sqlalchemy.ext.declarative import declarative_base
9 from sqlalchemy import *
10 import tensorflow as tf
11 import sys
12 import traceback
13 from os import listdir
14 from os.path import isfile, join
15
16 filepath = "/datadrive3/video-level-train"
17
18 # Initialize connection to to remote database. Make sure Azure VM is
      running
19 Base = declarative_base()
20 engine = create_engine("postgresql://usman:ayyaz@localhost:5432/postgres
      ")
21 metadata = MetaData(engine)  # MetaData object where newly defined Table
       objects are collected
22 engine.echo = False
23
24
25 class Youtube_Video(Base):
26     __tablename__ = 'train'
27     # tell SQLAlchemy the name of columns and their attributes:
28     video_id = Column(String, primary_key=True, nullable=False)
29     labels = Column(postgresql.ARRAY(Integer))
30     mean_rgb = Column(postgresql.ARRAY(Float))
31     mean_audio = Column(postgresql.ARRAY(Float))
```

```
32
33
34 def getFilenames(path):
35     files = []
36     for f in listdir(path):
37         if f.endswith(".tfrecord"):
38             files.append(join(path, f))
39     return files
40
41     # return [f for f in listdir(path) if isfile(join(path, f))]
42
43
44 if __name__ == '__main__':
45
46     Base.metadata.create_all(engine)  # creates all tables stored in the
       metadata
47     # Create the session
48     session = sessionmaker()
49     session.configure(bind=engine)
50     s = session()
51
52     tfrecord_files = getFilenames(filepath)
53     count = 0
54
55     for filename in tfrecord_files:
56         print
57         "####################"
58         print
59         "####" + filename + "####"
60         print
61         "####################"
62         print
63         count
64         count += 1
65         print
66
```

```
67          record_iterator = tf.python_io.tf_record_iterator(path=filename)
68          for string_record in record_iterator:
69              video = {}
70              frames = tf.train.Example()
71              frames.ParseFromString(string_record)
72              video["video_id"] = frames.features.feature["video_id"].
    bytes_list.value[0]
73              labels = str(frames.features.feature["labels"].int64_list.
    value)[1:-1]
74              labels = [int(i[:-1]) for i in labels.split(",")]
75              video["labels"] = labels
76
77              mean_rgb = str(frames.features.feature["mean_rgb"].
    float_list)
78              a = mean_rgb.split("\n")[:-1]
79              video["mean_rgb"] = [float(i.split(":")[1]) for i in a]
80
81              mean_audio = str(frames.features.feature["mean_audio"].
    float_list)
82              a = mean_audio.split("\n")[:-1]
83              video["mean_audio"] = [float(i.split(":")[1]) for i in a]
84
85              try:
86                  record = Youtube_Video(**{
87                      'video_id': video["video_id"],
88                      'labels': video["labels"],
89                      'mean_rgb': video["mean_rgb"],
90                      'mean_audio': video["mean_audio"]
91                  })
92                  s.add(record)  # Add all the records
93                  s.commit()  # Attempt to commit all the records
94              except:
95                  print
96                  "Exception in user code:"
97                  print
98                  '-' * 60
```

```
99              traceback.print_exc(file=sys.stdout)
100             print
101             '-' * 60
102             s.rollback()  # Rollback the changes on error
103         finally:
104             s.close()  # Close the connection
```

Query Class

```
1
2 import pandas as pd
3 from sqlalchemy import *
4 from exceptions import TsKeyError, FieldError
5
6
7 class TypeTime(object):
8     def __init__(self, tskey, time):
9         self._tskey = tskey
10        self._time = time
11        raise NotImplementedError
12
13    @property
14    def tskey(self):
15        return self._tskey
16
17    @property
18    def time(self):
19        return self._time
20
21    def __str__(self):
22        return '{0} ; {1}'.format(self.tskey, self.time)
23
24
25 class Query(object):
26    """
27    Interface to query schema
28    """
29    def __init__(self, tskeys, start_time_inc, end_time_inc):
30        """
31        query defined by (time, type) keys, start and end time (
    inclusive).
32        :param tskeys: set<TsKey> or collection
33                Time series keys
```

```
34          :param start_time_inc: np.datetime64
35              inclusive
36          :param end_time_inc: np.datetime64
37              inclusive
38          """
39          self._tskeys = frozenset(tskeys)
40          self._start_time_inc = start_time_inc
41          self._end_time_inc = end_time_inc
42
43      def execute(self,session,metadata):
44
45          Dataset = Table('datasets', metadata, autoload=True)
46          if len(self._tskeys) > 1:
47              raise NotImplementedError("not handling multiple queries yet
    ")
48
49          else:
50              # Check if Table exists
51              key = next(iter(self._tskeys))
52              if session.query(exists().where(Dataset.c.key == key.name)).
    scalar():
53                  print(key.name)
54                  timeseries = Table(key.name, metadata, autoload=True)
55                  if key.fields[0] == '*':
56                      key.fields = [c.name for c in timeseries.c]
57                      response = session.query(timeseries).filter(
58                          and_(timeseries.c.timestamp >= self.
    _start_time_inc,
59                               timeseries.c.timestamp <= self.
    _end_time_inc)).all()
60                  else:
61                      response = session.query(*[c for c in timeseries.c
    if c.name in key.fields]).filter(
62                          and_(timeseries.c.timestamp >= self.
    _start_time_inc,
63                               timeseries.c.timestamp <= self.
```

```
_end_time_inc)).all()
64
65                  table_columns = [c.name for c in timeseries.c]
66                  data = []
67
68                  for i in response:
69                      a = []
70                      for col_name in key.fields:
71                          if col_name in table_columns:
72                              a.append(getattr(i,col_name))
73                          else:
74                              raise FieldError("Field does not exist")
75                      data.append(a)
76
77                  result = pd.DataFrame(data=data, columns=key.fields)
78                  return result
79
80              else:
81                  raise TsKeyError("Key does not exist")
82
83      @property
84      def tskeys(self):
85          return self._tskeys
86
87      @property
88      def start_time_inc(self):
89          return self._start_time_inc
90
91      @property
92      def end_time_inc(self):
93          return self._end_time_inc
94
95      def __eq__(self, other):
96          """
97          :param other:
98          :return:
```

```
99          """
100         if other is None:
101             return False
102
103         if isinstance(other, self.__class__):
104             return self.__dict__ == other.__dict__
105
106         raise NotImplementedError("equals not implemented for " + other.
    __class__)
107
108     def __hash__(self):
109         return hash(self.tskeys) + 11*hash(self.start_time_inc) + 17*
    hash(self.end_time_inc)
110
111     def __contains__(self, typeTime):
112         """
113         check if the given (type, time) is contained w/i this
114         that is, typeTime.tskey in this.tskeys and time in [startTimeInc
    , endTimeInc)
115         :param typeTime: TypeTime
116         :return: true if typeTime.tskey in this.tskey and time in [
    startTimeInc, endTimeInc)
117         """
118         raise NotImplementedError
119         #return typeTime.tskey in self.tskeys \
120         #       and typeTime.time >= self.startTimeInc \
121         #       and typeTime.time < self.endTimeInc
122
123
124 class QuerySet(object):
125     """
126     Represents a set of IQuery's
127     """
128
129     def __init__(self, queries):
130         """
```

```
131         initialize this w/ the given queries
132         :param queries: set<IQuery>
133             IQuery objects
134         """
135         #_qset = queries
136         self._queries = frozenset(queries)
137
138
139     @property
140     def queries(self):
141         return self._queries
142
143
144     def __contains__(self, typetime):
145         """
146         Check if a given (tskey, time) is contained w/i this query set
147         :param typetime:
148         :return: boolean
149             true if time is covered by the given queries
150             false o/w
151         """
152         for q in self.queries:
153             if typetime in q:
154                 return True
155         return False
156
157
158     @staticmethod
159     def single(q):
160         """
161         Construct from single query
162         :param q: IQuery
163         """
164         return QuerySet(queries=frozenset([q]))
```

Bibliography

[1] MW Watson. Time series: Economic forecasting.

[2] G Peter Zhang. Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175, 2003.

[3] Kyoung-jae Kim. Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1):307–319, 2003.

[4] Roberto Salama. A regression testing framework for financial time-series databases: an effective combination of fitnesse, scala, and kdb/q. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 149–154. ACM, 2011.

[5] Gregory C Chow and An-loh Lin. Best linear unbiased interpolation, distribution, and extrapolation of time series by related series. *The review of Economics and Statistics*, pages 372–375, 1971.

[6] Mohsen Pourahmadi. Estimation and interpolation of missing values of a stationary time series. *Journal of Time Series Analysis*, 10(2):149–169, 1989.

[7] Maurice Bertram Priestley. Spectral analysis and time series. 1981.

[8] Peter M Robinson. Nonparametric estimators for time series. *Journal of Time Series Analysis*, 4(3):185–207, 1983.

[9] Wes McKinney, Josef Perktold, and Skipper Seabold. Time series analysis in python with statsmodels. *Jarrodmillman. Com*, pages 96–102, 2011.

[10] Statsmodels time series analysis - tsa. `http://www.statsmodels.org/stable/tsa.html`.

[11] Pyflux an open source time series library for the python programming language. `http://www.pyflux.com/garch-models/`.

[12] CRAN time series analysis in r. `https://cran.r-project.org/web/views/TimeSeries.html`.

[13] PySpark spark programming model to python. `https://spark.apache.org/docs/0.9.0/python-programming-guide.html`.

[14] P Dix. Influxdb: One year of influxdb and the road to 276 1.0, 2014.

[15] B Sigoure. Opentsdb scalable time series database (tsdb). *Stumble Upon*, 2012.

[16] Time Series Analysis. `https://www.math.kth.se/matstat/gru/sf2943/ts.pdf`.

[17] SQLAlchemy object relational mapper. `https://www.sqlalchemy.org/`.

[18] Timeseries forecasting using matrix completion. (in press).

[19] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

[20] tfRecord-SQL data plug. `https://github.com/UsmanAyyaz/tfRecord-SQL`.