LearnAir: toward Intelligent, Personal Air Quality Monitoring

David B. Ramsay

BSEE and BA, Case Western Reserve University (2010)

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning in partial fulfillment of the requirements for the degree of Master of Science at the Massachusetts Institute of Technology

September 2016

Author .....
MIT Media Lab
August 6, 2016

Certified by .....
Joseph A. Paradiso
Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .....
Pattie Maes
Academic Head
Program in Media Arts and Sciences

LearnAir: toward Intelligent, Personal Air Quality Monitoring

David B. Ramsay

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning
on August 6, 2016, in partial fulfillment of the requirements for
the degree of Master of Science

Abstract

Air pollution is responsible for 1/8 of deaths around the world. While the importance of air quality has led to a boom in inexpensive air sensors, studies have shown that the status quo of sparse, fixed sensors cannot accurately capture personal exposure levels of nearby populations. Especially in urban landscapes, pollutant concentrations can vary over just a few seconds or a few meters. Unfortunately, the portable monitors that are capable of accurately measuring these pollutants cost thousands of dollars.

That hasn't stopped a deluge of cheap, portable consumer devices from entering the market. These solutions frequently claim better accuracy, but universally fail under real-world validation. Instead of competing to *build* a more accurate sensor, we take the approach of trying to *predict* when we can trust the cheap sensor we have, based on ambient conditions and measurements.

Well-designed, sub-$100 sensors have recently started to perform with high precision and accuracy. While their fundamental operation is sound, these affordable sensors cannot incorporate costly, industry standard techniques for mitigating issues like cross-sensitivity, dynamic airflow, or high humidity. Fortunately, if the core principles of the device are robust, machine learning techniques should be able to predict systematic measurement failure based on a handful of related indicators. In this thesis, we test and demonstrate the potential for logistic regression machine learning techniques to predict and classify sensor measurements as 'correct' or 'incorrect' with high reliability. These techniques are also useful for quantifying sensor precision as well as cross-seasonal prediction strength.

After demonstrating the value of this approach, we implement a scalable database solution using a semantic web technology know as ChainAPI. The tools developed for this framework allow automatic learning algorithms to crawl through the database, access the most recent data, update their training model, and populate the database with the processed data for other crawling scripts to interact with. This backend has implications for air quality data storage, interaction, and exchange.

Finally, we build a portable, Bluetooth enabled air quality device that connects to ChainAPI through a mobile phone app, and takes advantage of the machine learning algorithms running in its backend. This device improves the reliability of sensor data compared with similar-cost systems.

The learnAir device empowers individuals to trust their personal air quality data, and provokes a dialog about sensor reliability in the citizen sensing community. Its novel database architecture promotes new ways of interacting with large, dynamic datasets, and new tools to characterize affordable sensors and devices. Finally, applied logistic regression algorithms assure the accuracy of cheap, distributed sensor data– creating a trusted way for researchers to collaborate with citizen scientists from around the world.

Thesis Supervisor: Joseph A. Paradiso
Title: Professor of Media Arts and Sciences

LearnAir: toward Intelligent, Personal Air Quality Monitoring

David B. Ramsay

The following person served as a reader for this thesis:

Thesis Reader.....

Steven Hamburg
Chief Scientist
Environmental Defense Fund

LearnAir: toward Intelligent, Personal Air Quality Monitoring

David B. Ramsay

The following person served as a reader for this thesis:

Thesis Reader.....

Ethan Zuckerman
Associate Professor of the Practice
Program in Media Arts and Sciences

Acknowledgments

Many selfless people advised, guided, and supported me during this thesis. I'm humbled and grateful to count them all as close friends and collaborators. I couldn't have done this without them.

First I'd like to thank my advisor, **Joe Paradiso**. He not only laid the groundwork for this thesis with his depth of expertise and strong connections; he has been an incredibly thoughtful, intelligent, and warm advisor. I will always be grateful for the chance he took on me, to join his lab and represent his vision in the world. I hope I might one day be half as sharp, capable, and kind.

The Enivornmental Defense Fund- particularly my reader **Dr. Steven Hamburg** and his collaborator **Millie Chu Baird**- have been instrumental in shaping the direction of this work, and providing great insight and invaluable community connections. They supported my Media Lab appointment and this work; it would not exist without them. I'm forever grateful not just for their funding, but for their active engagement from the very beginning.

Ethan Zuckerman has been a true mentor to me since this process began. I've never met a more helpful, brilliant, encouraging, and eloquent professor. He has been integral in shaping this project, but the most valuable things I've gleaned from him fall far beyond the bounds of this thesis and will stay with me well past its completion.

Ethan's group shares his spirit, and I'm indebted to the entire Civic Media Team. **Emilie Reiser** has been a brilliant collaborator and a wonderful friend throughout this process, investing countless hours helping and challenging me. **Don Blair** has also been incredibly warm, thoughtful, and constructive over many hours of conversation. The extended Civic family has been very generous with their time, and I'd like to thank all of them, particularly **Dave Mackintosh**, **Xiuli Wang**, and **Colin McCormick**.

I've relied on several experts to shape and inform my thinking about this project. In particular, I'm indebted to Safecast's **Sean Bonner** and **Pieter Franken** for their insight, which launched me into this project with a strong foundation. As I've continued, **Dr. Jesse Kroll** and **David Hagan** from MIT's Civil and Environmental Engineering Department have been very unselfish with their time and expertise. Their technical mastery of the field is truly inspiring.

I'm completely beholden to MassDEP- particularly **John Lane** and **Tom McGrath**- for allowing me 24/7 access to the EPA measurement site. This thesis wouldn't exist without their flexibility, and they've been delightful, responsive, and accomodating collaborators.

I'd also like to acknowledge my Responsive Environments family- particularly **Spencer Russell**, whose CHAIN work forms the basis for much of my contribution here (and who spent a tremendous amount of time helping me understand how to use it), and **Brian Mayton** for his technical insight and advice throughout the process. A big thanks goes to **Nan, Evan, Juliana, Asaf, Artem, Jie, Donald, Vasant, and Gershon** for useful, fun, and inspiring conversations along the way. It's a pleasure to work with people I admire so much.

Amna, Keira, and Linda- you three have kept me on track and been incredibly flexible and kind throughout the last two years. Thank you for the support, the smiles, and the gentle reminders. To my Boston friends - **Kristy, Chetan, Will, Nate, and Dylan** - thanks for putting up with me and keeping me sane throughout this process.

On a personal note, I'd like to thank the mentors that have invested in me, shaped me into who I am, and continue to challenge, guide, and inspire me. **Dan Gauger**, **Neal Lackritz**, **Ted Burke**, and **bunnie** - I can't overstate the impact you each have had on my life- as technical mentors certainly, but more importantly as confidants and role models. You inspire me to think creatively and make a difference through ambitious, high-quality work. You motivate me to live a more balanced life and approach the world with kindness and gratitude. You challenge me to re-examine my priorities, my goals, and my philosophies through your example. You've each made an indelible impact on my life, and I will continue to strive to follow in your example.

Most of all, I'm indebted to my wonderful and supportive family- my parents **Karen and Brad**, my sister **Tracy**, and the entire Benson/Ramsay clan. Thank you for believing in me, pushing me, and guiding me throughout the last 29 years. I admire you, I love you, and I owe you everything.

Contents

List of Figures

List of Tables

1. Introduction

Around the world, people are experiencing adverse health effects as a result of poor air quality. It is currently impossible for them to accurately track and understand their exposure using affordably-priced technology. While consumer monitoring solutions claim to address this disparity, time and again they are shown to be unreliable.

What if we could make cheap, personal air quality data trustworthy? What could individuals, communities, and research organizations achieve with this new data? How could we facilitate an open and collaborative ecosystem for sharing this data?

In 2014, the World Health Organization (WHO) revised previous estimates of air pollution related mortality– more than doubling them. [1] Shockingly, it is now estimated that one in eight total global deaths are the result of air pollution exposure, making air pollution the world's single largest environmental health risk. [1, 2] These revisions hint at the complexity of monitoring air pollution– the link from standard measurement techniques to personal exposure, and the further link from exposure to health, are still difficult to model. Failure to understand these relationships can undermine and obfuscate the health risks facing the global community. [1, 2, 3, 4, 5]

For the millions of people living in toxically polluted cities around the world, understanding their daily pollution exposure– and making informed changes to minimize it– could literally lengthen their lives. Affordable consumer devices have exploded over the last several years to address this need; however, none have proven reliable under real-world conditions. [6] Research continues to drive smaller and cheaper sensor technology, but the underlying physics and their associated manufacturing costs limit what is currently feasible.

The lack of reliable personal monitoring solutions is also a systemic problem. In many cities, political rhetoric and policy are in play as

citizen groups and research organizations mobilize around the issue. Local governments are installing distributed air quality networks– in some cases with citizen groups at the helm. Unfortunately, pervasive misinformation about the data quality of these new networks makes it easy for smart city initiatives to find themselves with solutions that yield limited benefits. At best this is a waste of time and money; at worst, it may precipitate unnecessary panic or ill-informed policy-making.

Proper education of politicians, citizen sensing groups, and the public is an important and difficult undertaking. In many cases, air quality rhetoric has out-paced its understanding, and the advocates working to correctly frame the discussion are playing from behind. Long-term, a market saturated with unreliable instruments could lead to disillusionment with the technology and friction surrounding the issue and its proponents.

While well-intentioned organizations figure out how to interact and guide community sensing initiatives to act correctly and responsibly, they also have a lot to gain from distributed, reliably-measured personal data. A breakthrough in personal sensing could unlock amazing insight into pollution modeling and epidemiology. Unfortunately, there are open questions in the environmental sensing world about how to store, interact with, characterize, and use these data. Even simple sharing of data amongst well informed organizations and researchers– without the confounding factor of unknown data quality– is an actively-debated issue. Differences in data labeling and data collection methodology make it difficult to agree upon a set of definitions and standards.

As we've seen, there are many serious challenges and much unrealized potential– for individuals, policy-makers, and organizations– that arise from the lack of trustworthy, affordable air quality monitoring approaches. Many researchers attempt to solve this problem directly by designing smaller and cheaper sensor technologies. Unfortunately, it is incredibly hard to improve the cost/reliability of a mature technology by simply re-optimizing the same core operating principles. Critically, there is also no agreed-upon standard for characterizing these device in real-world conditions. Many consumer devices unwittingly find themselves re-hashing the same core design principles and claiming improvement based on unrealistic, controlled laboratory tests.

In this thesis, we explore a novel approach to advancing the quality

of affordable, personal air quality data. Instead of fighting to *make* more reliable distributed sensors, our goal is to *predict* when a sensor is accurate and when it is not.

An affordable sensor may lose accuracy in predictable ways. An otherwise poor quality sensor may provide very trustworthy, repeatable data for a narrow set of climates, geographies, or seasons. Previously, there has been no systematic way to understand, classify, or predict these patterns. With this work, we test the limits of this approach and build a device that takes advantage of it. The learnAir system doesn't attempt to be more accurate than similar cost systems. It simply predicts when it is likely to be accurate, and when it is likely not to be.

The ramifications of such a design are numerous. If it succeeds, it will provide more accurate personal exposure information than comparable sensor systems– empowering individuals to more intelligently address any health concerns. It will also empower researchers and organizations to interact with data from historically untrustworthy sources. Furthermore, the algorithms underlying the learnAir system might be useful in matching existing sensors with climates and geographies where they will succeed– not only informing and improving the decision-making behind large-scale installations, but fundamentally altering the conversation to reflect its true complexity.

The learnAir system is designed in a scalable way, with an easy-to-use, open back-end. The structure allows sensors to compare themselves against one another, learn from one another, and automatically improve their learning models over time. As more sensors are added to the network, each benefits from the collective, shared data. For instance a sensor tested and deployed in the California summer will learn from the same type of sensor tested in a Boston winter– forming a more accurate and complete real-world model than either in isolation. This novel back-end design has powerful implications for the air quality community. It not only points to principles for sharing air quality data, but also stakes a claim in how researchers and engineers may utilize any distributed data set where quality is a significant variable.

The learnAir hardware is a portable, Bluetooth Low Energy (BLE) connected system that talks to its database through a smartphone app. In the cloud, its data is automatically compared to any nearby, higher quality EPA reference sensors, and the device learns to predict its own accuracy based on this comparison. LearnAir uses weather

data, as well as on-board sensors that measure temperature, humidity, light, wind, motion, and other pollutants, to predict its accuracy.

In other words, when the device is next to a higher quality sensor, it will automatically compare itself against the reference to see when its readings are consistent. If it discerns an inconsistent reading, it analyzes the weather and other ambient conditions to gauge whether there are any patterns that correlate with its inaccuracy. It also analyzes patterns that would suggest it is accurate.

Thus, every measurement learnAir makes comes with a prediction as to whether it is correct (as compared with a reference measurement method), as well as a probability that it has guessed its correctness accurately. This information is based on a shared data resource, which houses a constantly updating sensor model. This model takes into account all of the measurements ever reported to the system by that type of sensor.

In order to build this device and the supporting infrastructure, and in order to validate that a system of this type is useful, we pose the following preliminary questions:

(1) How well can machine learning algorithms predict the accuracy of an unreliable sensor if some information is available about the sensor's environment? What is the best approach to applying machine learning to this problem? Answering this question successfully has implications beyond air quality sensor networks, informing new topologies for high/low quality sensor interaction and system design for a range of networked systems.

(2) How can different systems of variable quality be supported using an intelligent, scalable, and open data structure? In this thesis we create many new tools for crawling and interacting with distributed data. The implications of this work again reach beyond air quality, in fields ranging from large-scale data sharing to the Internet of Things.

This thesis (1) answers the two foundational questions outlined above, and (2) uses the results to build a novel, deployable system. In it, we test whether machine learning can provide a more nuanced understanding of when and how inexpensive sensors succeed, to elevate their trustworthiness. We build a scalable database that supports scalable learning algorithms, where sensor data of any quality can co-exist seamlessly. Finally, we demonstrate a system that takes advantages of these techniques, evokes a fruitful dialog in the citizen sensing community, and puts that technology in the palm of your hand.

2. Background and Motivation

If I had an hour to solve a problem and my life depended on it, I'd spend the first 55 minutes determining the proper question to ask.

- Albert Einstein

There are many issues standing in the way of low-cost, portable, and effective air quality measurement. While the field is ripe for innovation, a thorough understanding of the state-of-the-art (and its limiting factors) is important before attempting to make progress. The following section is a distillation of advice from industry experts and thought leaders, who greatly informed the direction of this work.

Air Monitoring

Air Pollutants

Despite the complexities behind sparse air quality measurement and individual health, there is a plethora of evidence linking small particulate matter and other pollutants to serious negative health effects. [1, 2, 3, 4, 5]

Particulate matter is designated according to its size– PM2.5 are particles with 2.5 micron diameter or less, PM10 has a 10 micron diameter or less, and ultrafine particulate (UFP) are measured in nanometers (equivalent to PM1). These designations are made based on human physiology. Particles larger than 10 microns are typically filtered in the nose and throat; below this size, the particles are considered 'respirable'. Particles between 10 and 2.5 microns can usually penetrate into the lungs and settle, while particles less than 2.5 microns tend to pass into the alveoli and into the bloodstream. Nanoparti-

cles are so small that some can pass through cell membranes, and damage other organs throughout the body. Additionally, particulate deposition is different for these groups– PM10 may settle out of the air in hours, while the smaller and lighter particles generally stay in the air until they are washed out with precipitation.

Particulate size distribution is generally viewed as the sum of n log-normal distributions. [7] Nucleation by-products from engine combustion drives a positively-skewed, log-normal distribution centered at a few hundred nanometers. [8] Mechanically generated road dust on paved and unpaved roads generates a negatively-skewed log-normal particle distributions favoring 10 micron diameters. Pollen also has a negatively skewed log-normal distribution in the 10-100 micron range. [8] The greatest health risk is associated with the smallest diameter (often nucleation-based) particulate.

Specific gases have also been linked to health risk, and the United States Environmental Protection Agency (EPA) has set standards for five other pollutants– Lead, Nitrogen Oxides, Carbon Monoxide, Sulphur Dioxide, and Ozone. [9]

The main sources of lead exposure (automotive fuel and paint) have been heavily regulated over the past several decades in first world countries. The last industrial lead smelter in the United States shut down two years ago, and airborne lead exposure in the first world has largely been eliminated other than from old house paint (which can aerosolize as houses are retrofitted or torn down.) Airborne lead is still an issue in developing countries, however, especially near unregulated smelters that recycle car batteries.

The result of incomplete or high temperature automotive combustion, Nitrogen Oxides and Carbon Monoxide (as well as Black Carbon, a major constituent of PM2.5) are common pollutants in the urban setting. Since the sources of these pollutants are mobile, they often manifest with complex spatiotemporal dynamics, including temporary hotspots and/or highly localized areas of high exposure.

Sulfur Dioxide is the result of fossil fuel combustion, largely coal and heavy oil, and is thus detectable near power plants and other industrial operations.

Ozone at the earth's surface is typically the result of Volatile Organic Compounds (VOCs) reacting with Nitrogen Oxides in the presence of sunlight. Thus, while vehicle emissions seed the process, ozone

concentrations tend to be dependent on sunlight, and therefore more predictable and less dynamic than CO, NO2, and PM.

EPA standards are set at 75 parts per billion (ppb) average per hour for SO2, 100 ppb average per hour for NO2, 70 ppb average for 8 hours for Ozone, 9 parts per million (ppm) average for 8 hours for CO, and annual averages of 0.15 $\mu g/m^3$ for Pb, 12 $\mu g/m^3$ for PM2.5, and 35 $\mu g/m^3$ for PM10. [9] In the U.S. air pollutants have declined by over 60% since 1990, however 121 million people still live in counties were these standards are not met. [9] In the developing world these standards are rarely met in urban environments.

Sensor Technologies for Particulate Matter

There are many sensor modalities for pollution monitoring, with a handful considered reference grade. For measuring particulate matter, high quality installations will deploy either a Beta Attenuation Monitor (BAM) or a Tapered Element Oscillating Microbalance (TEOM) sensor. BAM sensors collect particulate on successive circular sections of a long filter that is spooled inside the device. Measurements are taken by simply analyzing the beta particle attenuation through the filter. TEOM sensors draw air through a filter so particulate deposits on the end of an oscillating cantilever (its resonant frequency is dependent on its mass). The more mass that is deposited on the filter, the lower the resonant frequency, which is measured and used to calculate very accurate particulate levels. Other methods typically include gravimetric techniques with filters that are analyzed in a lab environment. [10]

Figure 1: AlphaSense Optical PM2.5 Sensors

Optical methods for particulate sensing are particularly important in the mobile context due to their size and robustness. Since particulates are measured in $\mu g/m^3$, their accuracy depends on assumptions about airflow through the device, as well as the assumed statistics of particle size and mass distributions (which can change from location to location depending on the local sources and mixtures of pollutants).

Optical PM sensors typically have similar geometry- a narrowly focused IR beam is broken by the particles, and the scattered light is measured by an off-angle photodiode. For cheap sensors such as smoke alarms, airflow through the device is not tightly controlled (it may be driven by convection with a small heating element, but it

is never consistent), the optics are coarse, and the captured light is only a loose representation of particulate level. Better sensors use a more tightly focused beam, as well as a fan to control airflow, and can count pulses as the beam breaks– thus the length of time the beam is broken is proportional to particle size. The final and best type of optical technique uses Mie Scattering, which uses the intensity of scattered light on the photodiode to calculate particle size (a nonlinear monotonic function with particle size). [11] Using this information (while observing the duration of a particle in front of the beam), it is possible to calculate flow rate and corrected for any inconsistencies between the designed and observed air speed. Accurate optical variants include Condensation Particle Counters, in which particle size is increased in a predictable way by condensing vapor from a working fluid around the particles, making them easier to count. Electrical mobility sorting based on size/charge of particle in electrostatic field is also sometimes used in concert with optical techniques.

Inlets that protect these devices from wind, and thus help to create predictable airflow, are included on all professional-level equipment. These inlets typically include particle size selectivity. Most commonly, size selectivity is achieved using inertial techniques (i.e. impaction or cyclone filtering). Inlets are generally mounted in an upright position due to particle deposition (so gravity works with the inlet), and are sometimes heated to evaporate fog.

For cheap, small, or mobile applications, optical sensing is the dominant modality. On the high end (>$15k), handheld systems like the Grimm Enviro 11E have sophisticated inlets for size selectivity, advanced optics, and create sheathing airflow out of filtered air, which helps collimate incoming samples and clean the beam path. These professional-quality systems have been used in research studies to evaluate personal exposure, but they do not offer a viable option for distributed or personal applications.

On the affordable end of the spectrum there are many sensors. Unfortunately, independent tests have shown that none of these produce repeatable, accurate measurements in dynamic, real-world situations. [6] One in particular worth noting is the Shinyei PPD42NS, a $10 sensor that is widely used and cited as providing high quality results. [12, 13] It has a small heating element for inducing convective flow, and coarse optics. Unfortunately, its reputation is based on very controlled conditions. In EPA-conducted outdoor tests, the Shinyei demonstrated cross-sensitivity to variations in airflow, temperature,

and humidity. [9] The lack of clarity around the application space in which cheap optical sensors can be effective is unfortunate, and highlights the importance of testing these sensors in realistic mobile contexts.

The $300 Dylos seems to be the cheapest optical sensor with a strong linear relationship to Federal Reference Measurements (at <95% relative humidity levels) in independent tests. [6] It uses an IR laser and has a much larger and more sophisticated flow design than its cheaper counterparts. [14]

The $400 OPC-N2 is an optical particle sensor similar to the Dylos, but much smaller. It uses a fan to drive a fixed flow rate, and take advantage of Mie Scattering principles to correct for variations in flow rate through the device. AlphaSense, its manufacturer, has not released a full characterization of their design, but preliminary testing in environmental sensing groups at MIT and the South Coast Air Quality Management District (SCAQMD) have yielded mixed results– it appears that the OPC-N2 is incapable of sensing particles below a few hundred nm in diameter, which make up most of the mass concentration of PM2.5. [15, 16]

Since PM2.5 mass is largely made up of nucleation/combustion driven particulate, its core component follows a log-normal size distribution centered in the nm range. While measuring the log tail can provide some insight into the core of the distribution, tails from larger particulate like mechanically-created road dust or pollen (mostly 10-100micron) overlap in the critical 300nm-10micron range where the OPC-N2 is sensitive. This presents serious challenges inferring a relationship between what is actually measured in the 300nm-10micron range and PM2.5 levels without extra information.

Sensor Technologies for Gas Sensing

For sensing specific gases, many types of sensors are used. Among other techniques, spectroscopy, chromatography, and chemiluminescence are very common for professional applications. For mobile use, Alphasense sensors have emerged with a strong cost to performance ratio. Alphasense sells Photoionization Detection based sensors, which work by ionizing gas particles with UV light and sensing the generated current over a fixed voltage in contact with the air. They also sell Nondispersive IR sensors, a simple optical absorption

Figure 2: AlphaSense Electrochemical Gas Sensors

method.

For the specific types of gases we're interested in, electrochemical techniques are the primary low cost method on the market. The AlphaSense version is well-regarded, with ppb sensitivities and a clear failure condition (instead of the gradual drift you might expect as the sensor is depleted and dirtied). Electrochemical gas sensors are comprised of a working electrode, a reference electrode, and a counter electrode, all bathed in an electrolyte. The reference electrode is used to control the voltage at the working electrode, and keep it in a linear current/voltage regime. The working and counter electrodes promote inverse oxidation/reduction reactions, combining with the gas to produce free electrons, and then balancing that first reaction so as not to deplete or change the available reactants. The resulting current is proportional to the gas concentration, as long as corrections are applied for temperature, humidity, and pressure (and adequate time is allowed for 'warming up' once the reference electrode is powered on due to large inter-electrode capacitance). [17] These sensors are generally well-characterized under stable operating conditions, and have well understood cross-sensitivities and time-constants associated with their behavior.

While AlphaSense sensors can be purchased with calibration data, environmental sensing researchers at MIT have suggested that these calibrations are generally not accurate, and typically co-locate the sensors with a Federal Reference sensor for more rigorous calibration before deploying them elsewhere. [15]

Measurement Strategies and Complications

Historically, the standard measure of air quality has been a sparse network of fixed stations run by government agencies. These expensive and large stations require careful manual calibration every few weeks.

While these stations provide accurate data, studies have shown they either chronically underreport or have no correlation with the personal exposure of the citizens living near them. [18] Only with sophisticated modeling of elevation, geography, ambient conditions, wind velocity, and land use can these data be tied to exposure elsewhere in a city, and these models must be evaluated on a case-by-case and pollutant-by-pollutant basis.

New techniques are emerging to map and model a city using a small number of medium quality, mobile sensors. [19] Stationary, high quality sensors play an important role in calibrating these systems on-the-fly, but these methods have shown much better predictive power for mapping cities in higher spatial resolutions. However, their predictive power is still best on timescales of years and weeks, and starts to break down as they move towards days and hours.

Models on these timescales and resolutions are useful for understanding general trends in exposure for a city, as well as identifying and eliminating pollution sources, hotspots of high exposure, and issues with urban planning. However, even these mobile techniques for map generation are limited in their ability to predict personal exposure with high spatiotemporal resolution.

Personal exposure is so difficult to measure because pollutant concentrations can vary dynamically. For certain conditions researchers have modeled this complex behavior, and thanks to expensive portable sensors there have been several studies to corroborate their findings. One such dynamic system that has been analyzed extensively is the 'urban canyon'– a street with two tall buildings on either side that creates several interacting, swirling vortices [20]

Measurements have shown CO and UFP concentrations doubling on one side of the street relative to the other in an urban canyon, measured at the same time of day. [21] This variability has been demonstrated time and again– one study showed complex relationships between different pollutants measured in the center of the street versus the sidewalk. [22] Different corners of the same intersection can also vary tremendously. [23] Even walking roadside vs. building-side on the same sidewalk has been linked to significant differences in pollution exposure level. [24]

This is all to say that spatial variation is extremely high in some situations. Concentrations can change drastically over just a few meters. For accurate personal exposure monitoring, best practice is to sample air within 30cm of the mouth and nose. [18, 25] While some of these spatial phenomena may fit an urban canyon model, and some may be modeled accurately with standard dispersion models, pollutant levels in general are hard to predict with any single technique (especially to within a few meters). [26] Given the current state of air pollution modeling, it is extremely difficult to predict spatial variation at a scale relevant to personal health outcomes without direct measurements. [18, 25]

Temporal variation is equally difficult to monitor. [27, 28] Studies at traffic intersections have shown that regular, tenfold increases in pollutant concentration can occur over one second intervals. [29, 30] This staggering variation is averaged out even with the some of the best 'real-time' techniques– fifteen second integration could miss an entire elevated concentration event. Peak exposure levels may have important health implications, and transient events may account for the majority of urban exposure. [27, 28, 29, 30]

Given the tremendous spatiotemporal variation in pollution, fine-grained and distributed sampling in the lived environment seems to be the only viable path to accurate personal exposure data. In dynamic environments, personal and mobile sensors offer the most direct path to these data. While a dense fixed sensor network could provide similar insights, it would require many, intelligently placed devices to recreate the spatiotemporal resolution required to match the exposure estimates of a few personal sensors. Verifying the predictive radius of a fixed sensor in a complex, urban environment is also non-trivial. Personal, mobile sensing bypasses these complications.

With enough adoption, portable sensing could improve the collection and prediction of city-wide pollution mapping traditionally associated with fixed sensor installations. Eventually, distributed (and likely mobile) sensor data may even enable accurate path-based personal exposure modeling (statistically relevant on the order of meters and minutes), since the data is collected in the real microenvironments that dominate their exposure. Accounting for this otherwise highly specific spatiotemporal resolution would be difficult with any alternative method (day-level and 100 m^2 resolution is the best we see with predictive models right now). As sensors increases in accuracy and drops in cost, distributed sensing will usher in a new way of understanding the pollution landscape and our exposure to it.

Sensor Networks

Air Quality Sensor Networks

It is not uncommon to see publications describing cheap and portable smart-phone based air quality projects. [31, 32] In most cases, these publications focus on system design, and produce thought-provoking

work on the user-interface. [33] In cases where technologists explore new sensor design, it is rare they achieve compelling improvements. The past 20 years has seen a lot of incremental optimization in the most promising sensing modalities. Few research labs are positioned to push the state-of-the-art further by simply re-applying the same core physics without a new fundamental insight.

Outside of phone applications, true system-level research in the air quality space is uncommon. Most air quality networks use the same topology– one type of sensor device with standard, centralized data collection methods. The exception to this rule comes out of ETH Zurich's OpenSense project, where mobile sensors check their calibration as they pass higher-quality fixed sensors. [34] OpenSense has also pioneered methods for multi-hop mobile sensor calibration. Their work sets the standard for exploratory new air quality sensor network topologies.

In the consumer space, many projects and devices are being launched. Unfortunately, most devices do not stand up to scrutiny, and rarely do they offer technical innovation. None of these devices has succeeded at sustaining momentum with its adopters. SmartCitizen is an example– after a successful 2014 kickstarter with 600 backers and $68k raised, the SmartCitizen online network currently shows no active devices (despite 618 having been registered). [35] The constant barrage of 'new' monitoring devices– without accountability, without rigorous data-collection, and without real-world use-cases– saturates and dilutes consumer interest in these important issues.

Citizens aren't the only ones purchasing air quality sensor devices. Many cities are installing high-density pollution monitoring networks– in some cases, only later realizing that the data is not of sufficient quality to be of any use. London (quite publicly) recently released a network of GPS-tracked, tweeting pigeons with NO_2 sensor backpacks– while driven by a marketing firm as a (very successful) publicity campaign, no data has confirmed the value of this mediagenic approach. [36]

The EPA publicly states that distributed, cheap sensing technology will be a cornerstone of their future success. [37] As part of the effort to engage with active citizens and communities, the EPA measures and publishes data about low cost consumer devices. [6] Currently this is done by co-locating the consumer device with a Federal Reference Measurement (FRM) device outside for several months (usually through a change of seasons). This validation is not standardized

or rigorously defined in length, season, analysis, or in number of devices tested. Generally, the end result is a simple regression comparison (produced by hand), and a single designation for the sensor (i.e. 'good' or 'bad'). Other organizations do similar co-location experiments (like SCAQMD) in very different climates using different standards. [38]

Large Scale Data Sharing

The air quality research community is actively looking for solutions to facilitate inter-organization data-sharing, so that large scale collaboration can become more commonplace. They are also actively working to educate, involve, and benefit from the citizen sensing movement. There are many open questions around how to structure an ecosystem with variable quality data, how to define data standards, and how data should be hosted.

The most common solution for large-scale data sharing is to construct a centralized 'cloud' database with strict data standards and a strict ontology. Generally, users prepare their data to meet the standard, and then push their data to the database using some basic tools. As the most common structure, there are options that have library support for various file formats and hardware platforms. Examples include data.sparkfun.com (which integrates directly with Arduino shields) or plenar.io (which has easy csv upload and flexible data selection/access features).

ChainAPI

When it comes to robust solutions for large scale sensor networks that directly feed into a database, the possible options are less well-defined. An ideal ecosystem would allow large scale networks to interact seamlessly, while still allowing freedom in ontology and distributed hosting (similar to the World Wide Web). While there are several solutions starting to appear for Internet of Things applications (the so called 'Web of Things'), many are over-specified, and up until now the practical result has been for industry players to silo their hardware, their data, and their applications. Industry consortia are starting to address these problems, but the issues are currently unresolved.

ChainAPI [39, 40] is a thin, HAL- and JSON-based hypermedia solution for creating distributed, browsable data resources. It dictates enough structure to make resources easily linkable, new ontological relationships easily definable, and datasets easily accessible, searchable, and streamable. It leaves open the questions of ontology and backend database structure.

ChainAPI has been successfully used for a large-scale sensor ecological installation in Southern Massachusetts. [41] It serves as the backbone for many interesting data visualizations, audio mappings, sensor browsers, and future-looking tools. It provides extensible answers to many questions facing the world of large scale, distributed sensor installations and their associated data.

Machine Learning

Machine learning provides a way to design algorithms that learn and improve as more data is provided. These techniques have been applied to sensor data in a variety of forms. Examples range from predicting the number of people in a closed space by looking at changes in distributed sensor readings (i.e. temperature, humidity, light, and pressure), to predicting soil moisture based on remote sensing techniques (i.e. vegetation index and light backscatter). [42, 43] One notable research project used HVAC sensor data, both analytically and redundantly, to predict and verify when a sensor in the network has failed and automatically replace its unreliable data. [44]

Generally, these examples use supervised learning approaches with some form of cross-validation to validate success. While each uses a different core algorithm (and there is room to test and apply all of them to the air quality space), one worth mentioning is the logistic regression. Logistic regression is frequently used to predict engineering failure of products or systems. [45] It can be applied as a binary classifier (i.e., 'Is this sensor failing?'), as well as give a probability for each outcome (75% likey for 'yes' and 25% likely for 'no').

Motivation

Air pollution poses a risk to the health of people around the world. While standardized measurement techniques are highly accurate,

they are also extremely expensive and historically only limited to a number of stationary sites. These stationary sensors do not capture meaningful information about a citizen's personal exposure– the spatiotemporal variations of pollution concentration are too complex and too narrowly resolved to be captured with a single, distant sensor.

For accurate personal monitoring, wearable, mobile sensing offers a very attractive approach. Sensors do exist that can measure personal exposure, but there is a tight relationship between cost and accuracy– most are cheap and inaccurate. While elusive, portable and affordable sensing has the potential to offer powerful insights for both individuals and research organizations.

The lack of cheap solutions is not due to a lack of understanding. The core device physics of most sensors have been well-optimized over several decades, and the sophistication underlying reference level equipment is truly remarkable. Affordable sensors are starting to mirror the core principles of instrument-grade devices. Unfortunately, there are systematic failures in low-cost systems that are fundamental to the underlying sensor modality.

Optical sensors, for instance, require precision optics, heated inlets, flow control, and size-selective filtration. Solutions to these problems require extra power, extra size, or extra cost (and frequently all three). Addressing these problems likely results in a sensor that is neither cheap nor portable. Electrochemical gas sensors require a clean, precision doping process, a statistically-defined minimal exposed surface area, and compensation for flow rate, pressure, temperature, humidity, and electrical noise. The physics limit how small they can be, and the market limits how much cost can be driven out of the manufacturing process.

There are two main take-aways from this section– the first is that attempting to incrementally improve devices by re-exploring their core physics is a difficult proposition. The core physics are well-understood and companies have been optimizing them successfully for decades. **The first order problems with cheap sensors are *not* with the core device principles, but with well-understood failure modes (like flow control, fog, temperature dependence, or chemical cross-sensitivity).**

Since the core physics underlying cheap commercial sensors are approaching a very high quality, we can assume that inaccuracy is likely the result of systemic, predictable failures. If there was a

single point of failure (like an optical sensor that is reliable except when fog is present), it would be trivial to predict when the sensor data is reliable based on fog measurements. In real-world scenarios, however, multiple failure modes compound and obfuscate these underlying predictive patterns.

If cheap sensors are entering a quality regime where failure is increasingly predictable, it leads us to machine learning as a potentially powerful mechanism to improve reliability. Machine learning is perfectly suited to tease out these complicated underlying relationships. Instead of the common approach of *improving* sensor performance, the research suggests that *characterizing* and *predicting* sensor reliability in a nuanced way is novel, necessary, and potentially revolutionary.

In many cases, first-order predictions may work well to predict sensor accuracy. Gas sensors break in known ways and are specified for known operating ranges. Simple monitoring of temperature/humidity/pressure exposure, its air-flow, and gas sensors of cross-sensitive pollutants could provide extremely useful insight.

Second order insights are perhaps more interesting. For instance, the OPC-N2 particle counter is likely to be confounded by road dust or pollen. What if we could loosely approximate road dust exposure based on the user's location relative to a road, traffic patterns, the time of day, and the wind? What if we could predict O_3 measurement reliability based on the underlying drivers– NO_2, sunlight, and cloud cover? In this thesis, we explore both first-order and second-order insights using machine learning techniques.

Simple machine learning analysis could also provide an objective measure of sensor quality. How well machine learning can predict a sensor's behavior is a nuanced way to measure its repeatability. Sensors that fail randomly instead of predictably are inferior in design and construction.

For any of this to work, we need to compare our low-cost sensor against a high quality reference, so we can learn when it is providing spurious data, and what conditions may be indicative of that error. There is precedent for air quality network infrastructure that compares cheaper mobile sensors with a higher quality reference, but until now this has only been done as a basic calibration step. What we are proposing is the first self-correcting air quality system.

In order to build such a system, we require a backend solution that can automatically compare EPA data to a cheaper network installation. ChainAPI is well suited for this task, and in the process of building this infrastructure, we examine and address some of the biggest issues facing air quality data sharing, ecosystem building, and data interaction. We also create an infrastructure that may be used to automatically measure and characterize consumer device quality, in a very nuanced, climate- and geography- specific way.

Finally, we believe such a system has the ability to contribute to and provoke a more nuanced, informed dialog in the citizen sensing community. We propose the first device designed with the assumption that its data *won't be consistently reliable*, that we need to predict when it is useful and when it is not. Inherent in the design is the suggestion that the a sensor's success is complex, based on a variety of factors. This provocation could help inform and educate new users– cutting through noise instead of adding to it.

There are many interesting problems currently facing the air quality community. We believe a machine learning approach to predicting sensor accuracy could improve the reliability of cheap sensors, pushing the state-of-the-art forward. Validating data from affordable sensors would opening up a world of reliable, distributed data to the research community. In the process of testing this approach, we hope to build scalable solutions for data sharing and network interaction between affordable and expensive sensors. We are also engaging the citizen science and research communities with a new perspective on how to approach scalable, affordable sensing.

3. *Related Work*

In this section we consider past air quality work in the context of mobile or personal sensing. Additionally, we explore several of the cutting edge solutions for storing, manipulating, and displaying data of this type.

Air Quality

Safecast

Safecast [46] is an organization created in response to the Fukushima disaster to crowd-source radiation measurements using open-source GPS-tracked Geiger counters. They have succeeded at creating and publishing the most detailed radiation maps in the world, and are now looking to apply their expertise to air quality.

We are already closely linked with the Safecast team, which has been working on open data and open hardware air quality sensing for about a year. They've just finished an alpha version of a stationary air quality monitor.

Aclima/Google

Aclima [47] is a San Francisco startup specializing in environmental sensor networks. Last year, Google and Aclima started a mobile, car mounted air quality project. However, their algorithms and hardware have not been shared and they appear to want to control all the data they generate. The speculation is that they are using a quite expensive setup in the back of the car, slowly drawing in air over time.

Copenhagen Wheel Project

Out of the MIT SENSEable lab, the Copenhagen wheel [48] is mounted on a bike wheel to assist your pedaling, lock your bike, and capture data about air quality. The original project was unveiled in 2009 and subsequently licensed to a Cambridge Area startup (Superpedestrian). It is currently available for pre-order. Air quality is a tertiary feature of their project, and no hardware specs or details about what they are planning to monitor have been released.

OpenSense

An ETH based project, is the current world leader in mobile air quality sensor networks, with the first practical distributed mobile sensing platform using large devices mounted on the Zurich public tram system. [34] The OpenSense team has published many excellent articles that address some of the fundamental problems in low-cost, mobile air quality sensing. They've laid the groundwork for dealing with gas sensor calibration (particularly with multi-hop calibration techniques), as well as advancing the state of the art in Land Use Regression modeling based on sparse, real-time, mobile, distributed measurement. However, their resolution has only been tested up to 12 hour, 100m x 100m predictions (with accuracy much less than weekly/yearly predictions), and their sensors are large and run off of power from the trams they are mounted on. Their first attempt at truly personal sensing was a recent low-cost phone-enabled ozone sensor, which proved viable. [49] However, one of their assumptions for this device's success was that ozone has limited and predictable spatiotemporal variation (which is true, much less than other pollutants). Their preliminary tests with this sensor also revealed significant errors for high pollution levels when under airflow. [19, 50, 51]

TRUSS

Similar to OpenSense, the TRUSS (Tracking Risk with Ubiquitous Smart Sensing) project from the Media Lab's Responsive Environments group similarly blends high-power, fixed sensors with low-power, wearable nodes to provide real-time sensor fusion and insight. [52] In particular, this project examined and compared multiple air quality sensors to predict harmful conditions– however, the goal of

this work was to detect coarse, major risk instead of precise, ambient exposure.

Cheap Sensor Development and Testing Groups

There are organizations (such as South Coast Air Quality Management District [SCAQMD], the UK's National Physical Laboratory, and the EPA) that are testing and characterizing cheap optical PM sensors in real-world scenarios. There is also an active research community around developing MEMS PM sensors (using tech like FBAR, or Film Bulk Acoustic Resonators). These groups are essential for independently testing the claims of new products, and provide important insight into the trends with respect to the success and failure of new sensor technology.

Other

There are many other projects that appear in the air quality space, each with some unique take on affordable, distributed air quality. Most of the following projects are successful in one or more aspects of their system design, but do not innovate with their core sensor physics or algorithm.

These projects include MIT's stationary/expensive Clarity sensing project, [53] the independent and nicely designed tricorder sensor platform, [54] the innovative Propeller Health project that extrapolates pollution data from asthmatic inhaler usage, [55] the open source and mobile UPOD project, [56] the mobile Italian uSense project, [57] the 'wearable' AirBeam kickstarter project, [58] the Boston based startup Elm that translates their distributed air quality data into usable advice (and has an open invitation on their site to hook your sensor into their network), [59] the Israeli BreezoMeter startup that extrapolates street-level/high resolution air quality data (presumably from wind patterns and known reference data), [60] the London-based company cleanSpace that provides air quality maps and incentivizes air quality friendly commuting, [61] the startups Clarity and Tzoa that are selling versions of the 'world's smallest' PM2.5 wearable, [62, 63] and the startup uHoo which is building a home air quality monitor. [64] Other slightly less interesting, but still notable, include a handful of other hardware/commercial projects (the Air Quality Egg, [65] CMU's Speck, [66] Atmotube, [67] Air-

boxLab, [68] SmartCitizen [35]) and independent research like the French citizenAir project [69] and DIY Public Labs work. [70] New projects in this space appear on a regular basis.

Data Sharing Solutions

There are many options for data-sharing, particularly when exploring options that have a centralized entrypoint. Open source tools like Django [71] allow system administrators to easily spin up their own database solutions and create front-end tools, while services like plenar.io offer simple, managed solutions for data upload, display, and download.

An interesting open-source solution is Apache Hadoop, [72] which is a database back-end that can handle distributed storage and distributed processing for very large datasets. It can run on Amazon Web Services or Google Cloud, and has been used by many large companies. While interesting, it is not designed for easy input/output of data, but instead for managing and parallelizing computation on large datasets using Java (where everything is done in the cloud). This solution is likely over-technical and over-specified for the air quality use case.

The Semantic Web - RDF, HAL, and JSON-LD

An alternative, simple, distributed solution comes in the form of the semantic web. Instead of thinking of our data as separate from our devices– where researchers must pull data by hand, process it, and then upload it to a repository– it makes more sense to take an 'Internet of Things' approach. The advantages of this system are many– it provides transparency to the measurement technique (data is associated with the device, and metadata about the device, that uploaded it) as well as to the processing steps (data is uploaded in a raw form, and processed in the web). It also provides an intuitive, physical hierarchy for organizing and linking device and sensor data resources together. With proper system design, the researcher should not have to manually upload anything– they should be able to automatically pull all of the latest data (including automatically calibrated/processed data) from the database without ever manually pushing it there in the first place.

This type of sensor network design is not new. Most large sensor network installations automatically push their data up to a central server. By adding a semantic web layer on top of these servers, it is simple to connect devices and sensors in a much larger 'Web of Things'. Many standards for achieving this have been proposed over recent years.

Resource Description Format (RDF) is one of the most noteworthy foundations in semantic web thinking. [73] It represents a data model specification– every web resource (for instance, a device or a sensor object stored at a given URI)– has defined relationships through 'triples' of the form subject-predicate-object. An example would be 'device 1 includes sensor 5', which would be represented with URIs (http://device/1, http://relation/includes, http://sensor/5). This creates a labeled, directed multi-graph data model. It has been adapted to many standards and syntaxes like XML, and extended in standards like OWL (Web Ontology Language) or Turtle (Terse RDF Triple Language). The syntactic extension of this format in JSON – the current lingua franca of web data– is known as JSON-LD, and has been advocated for by Tim Berners-Lee (inventor of the World Wide Web). [74]

HAL, or Hypertext Application Language, is another hypermedia semantic web data model with XML and JSON formats. [75] It similarly defines relations to other resources using a shared URI, so ontologies can be easily added and extended. It is simpler than JSON-LD, with each resource limited to its attributes and its external link relations. HAL supports simple embedded resources and URI prefixing.

IoTivity and AllJoyn

Industry consortia are starting to take steps to build their own 'Web of Things' using the internet backbone. The two largest are IoTivity– a project by the Linux Foundation and Samsung– and Qualcomm's AllJoyn. [76, 77] They support low level protocols like CoAP, large APIs for interfacing with connected devices, and end-to-end device discovery and connection solutions. While these tools may gain prominence over the next several years, for now they are in their infancy, and looking to address more fundamental infrastructure/-connectivity problems (focusing more on real-time queries and connectivity rather than data storage, management, and sharing).

ChainAPI and TidMarsh

ChainAPI [39, 40] is a standard for data sharing based on HAL and JSON developed in the Responsive Environments group at the MIT Media Lab. It does not attempt to specify end-to-end connectivity, nor does it force any specific implementation or ontology– it simply wraps the HAL semantic web specification with common-sense design principles to make interoperability and data access simple and intuitive. This level of specificity enables a very low barrier to entry and very flexible use, while still providing enough structure to support a large, coherent ecosystem.

ChainAPI has been tested and used in the MIT Media Lab's TidMarsh project to store and expose ecological data from hundreds of local sensors over dozens of devices. This implementation includes a browsable web front-end, and several forward looking applications that take advantage of ChainAPI's streamability– a live, constantly updating virtual representation of the sensed environment is available at tidmarsh.media.mit.edu. Electronic music compositions have been written that stream and use live data from the Marsh. [78] Several elaborate data visualization scripts have also been written on top of ChainAPI. [79]

ChainAPI has a low barrier to entry, an extensible ontology, provides easy access to streamable data, and can quickly scale in a distributed manner. It is as simple as possible without sacrificing functionality.

4. Overview of Design and Contributions

The are three main goals with the learnAir project. The first is (1) to evaluate the usefulness and feasibility of applying machine learning algorithms to air quality sensor networks. Specifically, this means testing whether we can predict when a low-cost sensor is giving reliable data based on the conditions under which it is measuring. The second goal is (2) to prototype a data solution that addresses the needs of the air quality sensing community– particularly, the creation of an ecosystem that supports the needs of participants from high-end research facilities to lower-accuracy citizen projects– and allows seamless, easy interaction between them. It also must support the machine learning algorithms validated as part of our first goal in a scalable, automatic way (leveraging on-board, contextual sensors like temperature to help ground the data from its own air quality sensors). Finally, we want (3) to prototype an actual, handheld system that uses the algorithms from (1) and the database from (2) to realize a deployable, useful mobile sensor system that demonstrates the concepts and ideas put forth in this thesis. We believe this device will serve as a powerful rhetorical tool and push the dialog forward in citizen sensing communities. We also believe this system will give improved data reliability and as a result a more accurate estimate of personal pollution exposure compared to comparably priced and spec'ed systems.

The final system will enable a sensor to learn about itself when it is near a higher quality reference. It will apply that knowledge to new measurements made under new conditions, even as it moves away from the reference system. Every measurement the system makes will be accompanied by a simple prediction– was this measurement accurate or not? Every prediction will also come with an estimate of certainty– how confident are we this prediction is accurate? As more sensors of the same make and model are added to the network, predictions will improve and prior predictions can be revised.

Machine Learning Validation

Sensor technology– especially in the air quality space– is getting cheaper and more reliable. For the first time, well-respected sensors are starting to appear in the $100 range. However, there are limitations on what is possible with small, affordable technology. For example, electrochemical gas sensors break down in known ways– their reactions are temperature, pressure, and humidity dependent, in some cases they are cross-sensitive to other gases, and their reactions have time-constants and noise susceptibility based on electrode size and exposure area. Optical particulate sensors are frequently designed to mitigate external effects by applying (expensive) precise air flow control, heated inlets to eliminate fog, size-selective particle filtering, and advanced optics.

Sensors that break down in systemic ways may provide very reliable information under certain conditions. While it may be obvious if a sensor has a single, well-understood failure mode (for instance, an optical sensor that is very reliable unless there is fog), compounded failure modes are more common and more difficult to infer. By applying machine learning to this problem, we can automatically characterize which and how strongly underlying features are predictive of systematic errors.

Machine learning has a strong likelihood for providing insight into sensor reliability assuming the sensors are not spurious: that their failures come predictably. This assumption has likely not been true in the past, but the evidence suggests that we've entered into a new era of cheap air quality sensing– where some devices are based on trustworthy, strong designs, but cost constraints have prevented the inclusion of a controlled environment typical for instrument level devices. In these cases, we can measure the conditions instead of controlling them, and make an educated prediction about the reliability of any measurement.

To test this theory, we built a stationary device that can measure ambient conditions (temperature, light, humidity, wind) and included a range of cheap air quality sensors for CO, NO_2, O_3, and particulate. This sensor is called 'learnAir V1'. This sensor was installed for 2 months at a Mass Department of Environmental Protection (MassDEP) measurement site next to the inlet for reference EPA measurement equipment. The data collected from our measurements were compared to the EPA reference data as a 'ground truth' refer-

ence, to characterize when the sensors were reading accurately and when they weren't. We then used machine learning techniques to predict when the sensor was giving accurate readings or not. Using cross-validation techniques (splitting our collected data into a training set and a testing set), it is possible to characterize how well our algorithms predict sensor accuracy. See Chapter 5 for a description of the device and MassDEP reference hardware, and Chapter 7 for an in-depth analysis of the machine learning techniques and cross-validation results.

ChainAPI Instance and Tools

The air quality research community is actively working to mitigate the complications involved with inter-organization data-sharing. Furthermore, they are interested and actively engaged in questions around the citizen sensing movement– how should they inform and involve citizens in the air quality monitoring community? How should they validate cheap consumer devices? Is there a research use for the lower-quality data gathered from these networks, and how can they access and interact with it?

Another contribution of this thesis is to build and adapt an example of ChainAPI– a hypermedia data-sharing framework– for use with air quality data. ChainAPI offers many interesting advantages for sensor deployments as a thin, distributed hypermedia layer for linking data resources. It provides unique advantages for a diverse, distributed ecosystem where different sub-communities can co-exist.

In addition to creating a new ontology for air quality resources and building a development ChainAPI server, several new tools are necessary to interact with the ChainAPI air quality ecosystem. The infrastructure built for this thesis provides automatic resource discovery and dataset creation– so researchers can automatically find and interact with new datasets based on the type of data they are interested in working with, without *a priori* knowledge of where to look for that data. Its tools provides separation of concerns, so that raw data and data processing scripts are separated, instead of pre-processing occurring in an opaque way before data upload. Its processing scripts– that crawl through ChainAPI and update data– offer transparency to data quality and data manipulation.

This topology allows experts with certain sensors or devices to create,

own, and update the processing elements for their technologies, and provide automatic, high-quality processing to novice users. It provides the opportunity to create scripts that crawl through datasets looking for anomalies, out-of-spec operating characteristics, or out-of-service parts, and warn the device owners or data users. Finally, it offers the ability to create learning algorithms that automatically improve as more data is added to the ChainAPI ecosystem.

This contribution is an example of how ChainAPI could be adapted for air quality data sharing. It includes a new ontology based on the needs of the air quality community, and a large suite of tools that make ChainAPI a scalable, powerful tool for dynamic, growing data ecosystems. It also forms the basis for a truly automatic implementation of our machine learning algorithms. See Chapter 6 for an in depth discussion of the contributions around ChainAPI.

A Provocative Example

Based on the infrastructure developed with ChainAPI and the algorithms tested at the EPA site, the final goal of the project is to build a fully functional, portable device that integrates all of these features. There are several motivations for this: (1) as a rhetorical tool that can engage the citizen sensing community in a dialog about sensor data quality and validation, (2) as a means for deploying successful results from the previous sections as a best-in-class, trustworthy, manufacturable device, and (3) as a tool and platform for future testing of mobile use cases without having to do extensive data processing and manipulation by hand. Once this system is built, the machine learning results and comparisons can be generated easily and automatically as long as the data is available in the ChainAPI ecosystem.

Two portable prototypes were designed and built as part of this thesis– called 'learnAir V2' and 'learnAir V3'. Both are battery powered, hand-held devices that integrate AlphaSense electrochemical and particulate sensors. Both connect to a smartphone over BLE and push their data up to ChainAPI using GPS data from the phone application. Both monitor ambient conditions of their measurements (vibration, temperature, humidity, light, wind) to help predict their accuracy.

The two versions are very similar– the main difference is the core microcontroller. Version 2 is based on the Atmel ATmega32u4, and

was programmed using the Arduino environment. Version 3 is based on the lower-power, fully-featured STMicro STM32L152, which requires a more complicated build process, and comes with less library support. There are cost and power savings associated with Version 3 (making it more 'production-worthy'), but functionally they are nearly identical solutions. See Chapter 5 for an in-depth discussion of this hardware.

5. Hardware Design and Analysis

The vision of learnAir is to create a high-quality, affordable, and portable air quality monitor that (1) measures several pollutants, (2) measures ambient conditions like temperature, humidity, and light level, and (3) connects to a smartphone application that can display results, can send timestamp and geotagged data to the cloud, and can receive and display a prediction of measurement certainty based on machine learning applied to the data in the cloud.

Three learnAir devices have been built and tested. The first (learnAir V1) is not portable, internet-connected, or battery powered– it was solely used to collect sensor data that could be compared against higher quality MassDEP data for testing the feasibility and usefulness of predictive machine learning techniques. This device was installed on a MassDEP (Department of Environmental Protection) monitoring site for 59 days– from April 15th to June 13th 2016.

The second and third devices were designed to take the information gleaned from learnAir V1 and put it in a cheap, portable, smartphone connected package. The main circuit boards for LearnAir V2 and V3 were designed to match the size of the AlphaSense Analog Front End conditioning board– a reasonably priced production board with three electrochemical gas sensors that has gained a strong reputation in the pollution sensing community. These boards are BLE-enabled, and connect to two custom daughter boards for monitoring temperature, humidity, light level, UV exposure, and 3-axis wind speed.

The two boards share power circuitry and most peripherals– their central distinction is the main microcontroller. LearnAir V2 is a fully functioning board based on the Atmel ATmega32u4, which requires an external RTC (real-time clock). Firmware was written using a modified version of the Arduino codebase. The third version is based

on the STM32L152– a more sophisticated, production-level microprocessor, with more optimization for low-power states, more SD card read/write support features, and an onboard RTC. This third revision was also designed with a significantly smaller, soon-to-be available MEMS pressure sensor, which could reduce the volume occupied by the wind sensing module 50-fold (a large space savings from the current, large volume of 20mm x 60mm x 22mm).

The boards connect to a cross-platform smartphone application written in Javascript using Phonegap, a library that will compile javascript as webviews into iOS and Android applications. It has plugins to access the phone GPS, and a simple D3.js library was used for plotting. ChainAPI- our back-end solution- supports websocket connections and a subscription model to push data to the cloud and receive the latest predictions.

LearnAir Version 1

LearnAir V1 was created to collect data with a variety of cheaper air quality sensors at a MassDEP monitoring site, to test our ability to predict a sensor's accuracy with machine learning by comparing it to a high quality reference. The final box is 200mm x 120mm x 75mm (8 x 5 x 3in), and houses two main subsystems.

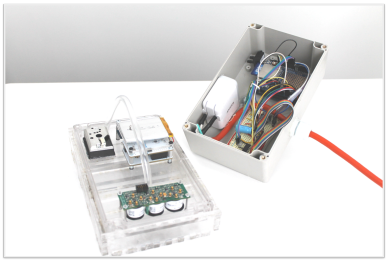


Figure 3: LearnAir Sensor installed at MassDEP site, opened

The first sub-system is an off-the-shelf air quality monitoring system called the SmartCitizen Kit (frequently abbreviated SCK). [35] The SmartCitizen Kit is Arduino-based (using the ATmega32u4), so custom code can easily be written and applied to the hardware. This system was used in an offline data-logging mode, with raw data streams stored to the onboard micro-SD card for later retrieval.

The SmartCitizen Kit includes several important sensors for our machine learning test. It has a **DS1307** real-time clock for timestamping data and sample timing, a **MicroSD slot** for saving CSV files, a ROHM **BH1730FVC** I2C light sensor to monitor ambient light levels, a PUI **POM-3044P-R** electret microphone for monitoring ambient noise level, and a Sensirion **SHT21** I2C temperature and humidity sensor. Most importantly, it has an E2V **MiCS-4514** CO and NO_2 sensor. This is a $10, MEMS sensor– one of the cheapest air quality sensors available. It works on a Reduction/Oxidation principle, and has small internal heating elements. It claims a 1-1000 ppm measurement range for CO and a 0.05-10ppm measurement range for NO_2,

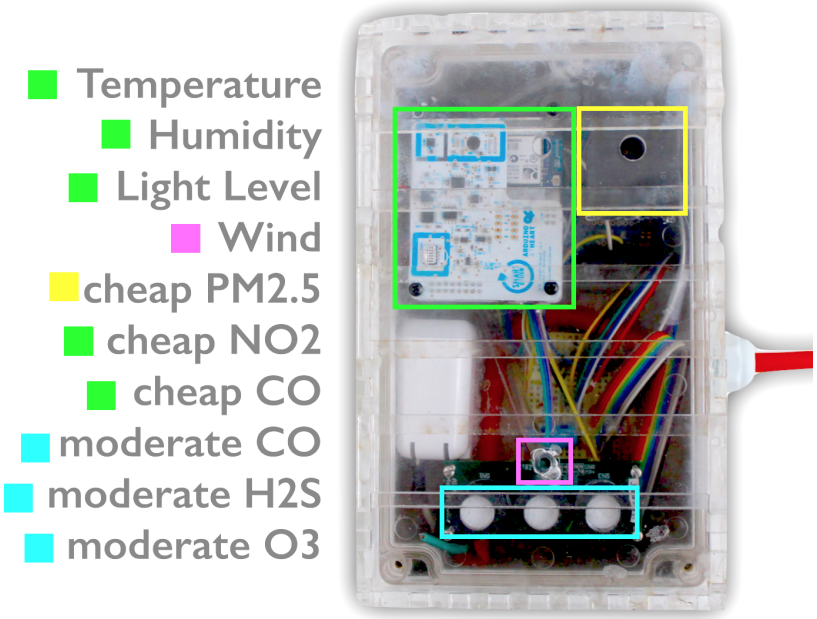


Figure 4: LearnAir Sensor installed at MassDEP site

and omits any information about reaction speed or cross-sensitivity in the datasheet.

The SmartCitizen Kit was mounted to the front of the case, with a small hole to expose the relevant sensors to the air. Every hole in the case was gasketed with silicone, and the front of the case was mounted face down to minimize rain exposure. Furthermore, a clear-acrylic, overlapping, slotted cover was designed to further protect the exposed sensing elements from rain and direct exposure.

Despite the effort to protect the circuitry, the SmartCitizen Kit corroded severely in our first outdoor test. A new kit was installed, this time with a layer of conformal coating added to prevent corrosion (Figure 5). This addition prevented further corrosion for the remainder of the two month installation.

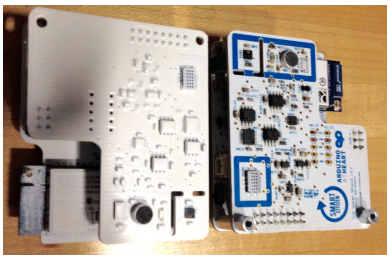


Figure 5: Corroded SmartCitizen Kit on the right, Conformal-coated new kit on the left. Relevant sensors on the new kit were taped off before coating to prevent contamination

Besides the SmartCitizen Kit, a custom Arduino-based sub-system is part of the platform. This system is based on an Arduino Leonardo board (Atmel ATmega32u4) with an Adafruit data-logging shield (which includes an **SD card slot** and a Maxim **DS1307** RTC for timestamping– the same RTC as the SmartCitizen).

Three main sensor sub-systems were connected to this Arduino using

a secondary prototyping board. First, a $10 Sharp **GP2Y1010AU0F** Optical Dust sensor was connected. This sensor requires an external charging circuit to store energy for a narrow IR light pulse that is used to detect scattering. These sensors have a reasonable reputation, and have shown good correlation with better quality sensors under controlled conditions indoors. [12, 13] In real-world situations, they are unreliable.

Secondly, an **AlphaSense Analog Front End Board** was connected. This board supports three AlphaSense electrochemical gas sensors, which each have a Working and Auxiliary Electrode measurement. Additionally, this board provides an onboard temperature measurement for sensor calibration. All of these signals were multiplexed through an external TI **CD54HC4051e** multiplexer before connection to the Arduino 10-bit ADC. AlphaSense sensors are well-reputed, mid-level sensors– they typically cost $60-100 each, with the support circuit adding an additional $150 in cost. These sensors have t_{90} response times of 10 ppm in 20 seconds for CO, 1 ppm in 30 seconds for NO2, and 100 ppb in 15 seconds for O3. Cross-sensitivity of the O3 sensors to NO2 is quite high (70-120% measured with NO2 levels of 5ppm), as well as vice versa. (The NO2 sensor picks up 30-60% of O3 at 100ppm.) Other cross-sensitivities are well-documented and much smaller. These sensors sport very low monitoring thresholds, well-characterized calibrations for electrolyte depletion and temperature dependence, and specified operating ranges for pressure, temperature, and humidity.

Finally, an Omron **D6F-PH** differential pressure sensor was included as a cheap, experimental way to measure airflow and wind. This pressure sensor has two outlets- one was left vented into the box, and one was connected through a tube to the surface of the device. Airflow over the top of the device creates a measurable pressure differential. This sensor runs an I2C interface at 3.3V (which is incompatible with the 5V Arduino), so an external BSS138 level shifter from Adafruit was required to properly interface with the sensor. All of these sensors were mounted to the front of the case, and gasketed to prevent unintended air exposure.

The final system is shown in Figures 3 and 4. An extension cord was spliced and soldered to a dual port USB charger, with powered both systems off of 5V USB. This extension cord was inserted through a cable gland in the side of the box. A metal mounting bracket extends from the back of the device. Both sub-systems were configured to sample their sensors every 30 seconds.

MassDEP Site

Thanks to the generosity of the Massachusetts Department of Environmental Protection, I was given full access to their Roxbury Monitoring Site, allowed to co-locate the learnAir V1 sensor with their sensing inlet (approximately 3 feet away), and provided (normally unpublished) high time-resolution data. The Roxbury monitoring site is the only one in the greater Boston area that has the capability to monitor particulate levels (through hourly Beta Attenuation Monitoring [BAM] measurements of black carbon), as well as minute-resolution data for trace gases (CO, NO, NO2, and O3). We were also provided with high quality, minute-resolved windspeed and wind direction data, which we used to analyze our experimental differential pressure wind sensor and included as a training feature in our machine learning data.

Figure 6: A picture of the Roxbury MassDEP measurement site where the LearnAir sensor was installed

The data provided by MassDEP is monitored weekly for quality assurance. Each piece of sensor technology is in the $10-100k range. The EPA specifies Federal Reference Method (FRM) devices (devices that are accepted as the reliable standard for research use and to compare other devices against), as well as Federal Equivalence Method (FEM) devices, which may use other techniques but are acceptable replacements for FRM techniques. All of these reference devices include FRM or FEM certification.

Black Carbon measurements were done with a Teledyne Model 633 Aethalometer sensor system which operates similar to the BAM style measurement. This type of measurement draws air through filter paper for a period of time, and then irradiates it with UV and IR light to see how well the captured particulate attenuates it. This means that– though the measurement is incredibly robust and reliable– the time-scale for measurements is on the order of an hour, and is an average of the accumulation over that period. For the system we were using, a '10 a.m.' reading is exposed to air from 10:00 to 10:50 a.m., and measured between 10:50 and 11a.m., before the next reading begins. Measurements are reported as $\mu g/m^3$.

Wind speed and direction data was captured on a minute time-scale with a high-quality, vane style Met One 50.5H Sonic anemometer. Trustworthy wind sensing data at these price points is non-controversial. Measurements are reported in approach angle (degrees) for the direction and m/s for speed.

Figure 7: LearnAir Sensor installed at the MassDEP site, close-up

For gaseous pollution measurements, all of the reference equipment comes from Teledyne Advanced Pollutant Instruments. The system is fed by an expensive, size-selective inlet with precise flow control. Air is actively pulled through the system at a known, fixed rate. Cross-sensitivity is not an issue for any of these devices.

For NO and NO2 measurements, the Teledyne Model 200E Chemiluminescence Sensor is used. NO is measured by exposing the gas to O3 and measuring chemiluminescence, and a catalytic-reactive converter then converts NO2 to NO and repeats the measurement. It has a precision of 0.5% of its reading, and a t_{95} of 60 seconds for its full operating range of 20 ppm, at a 0.5 L/min flow rate. It is capable of reporting the average over a sample period or the instantaneous value. It includes an adaptive filter that averages 42 samples over 5.6

minutes by default, unless it detects a rapid change in concentration (comparing an instantaneous reading to the long filter average), in which case it switches to a short-term, 6 sample, 48 second average.

For O_3 measurements, the FEM Model T400 UV Absorption system is used. It has a 0.5 ppb sensitivity, and a t_{95} of 20 seconds for its full operating range of 10ppm at a 800cc/min flow rate. This system works on the Beer-Lambert law, alternately comparing the absorption of a stream the sampled air with an O_3 filtered stream every 3 seconds using UV light (and correcting for temperature and pressure of the gas). It is capable of reporting the average over a sample period or the instantaneous value. Its adaptive filter averages 32 samples over 96 seconds by default, switching to a short-term, 6 sample, 18 second average with rapid changes in concentration.

For CO, the FEM Model 300EU Gas Filter Correlation system is used. It has a 0.5% precision, and a t_{95} of 30 seconds for its full operating range of 100ppm, at a 1.8 L/min flow rate It similarly uses the Beer-Lambert law and IR light to compare a scrubbed sample with the untouched air. It is also capable of reporting the average over a sample period or the instantaneous value. Its adaptive filter averages 750 samples over 150 seconds by default, switching to a short-term, 48 sample, 10 second average with rapid changes in concentration.

We installed the learnAir system face-down on the roof railing of the main air monitoring building, about three feet from the reference sensor inlet, as seen in Figure 8. Our sensor and the inlet both face downward, however the Federal reference system has active airflow. These inlets are mounted approximately 12 feet in the air and about 30-40 feet from the street.

Figure 8: LearnAir Sensor (box on left) installation next to MassDEP inlet (top of pole on right)

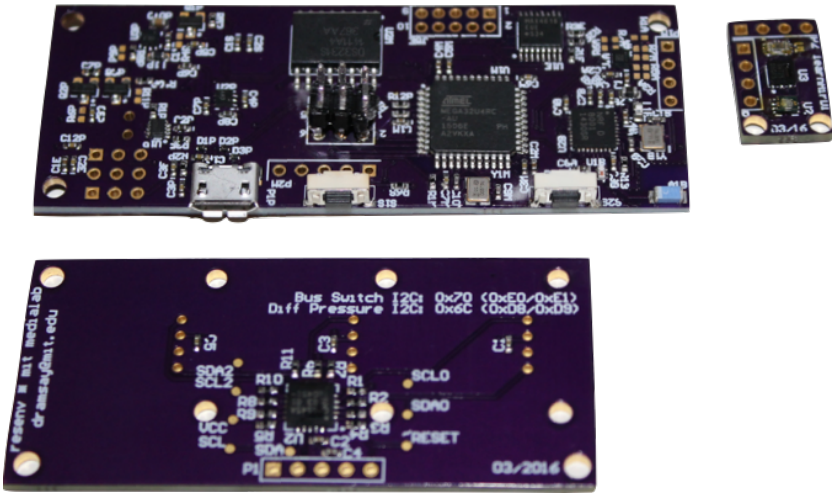


Figure 9: Main and daughter boards of learnAir V2.0

LearnAir Version 2

While the first hardware device we built was designed to collect data for testing and validation of learning algorithms, it does not represent a scalable, portable, cheap solution. For that, we designed learnAir V2.

LearnAir V2 was created to be inexpensive, handheld, and portable. It is designed to measure ambient conditions– just like learnAir V1– while still collecting the relevant air quality data. It is battery-powered and smartphone connected, so that data can be seamlessly GPS-tagged, viewed in real-time, and sent to the cloud. The outline of the internal circuit board was designed to mate with the AlphaSense AFE board. AlphaSense sensors come well recommended in the air quality sensing community.

The circuit design for LearnAir V2 is shown in Figure 9. It consists of a main board, designed around the Atmel ATmega32u4, and two daughter boards. The main board sports a **micro-SD card** slot for storing data, a Nordic **nrf8001** with a chip antennae for Bluetooth Low Energy communication with a phone, a **microUSB** connector for charging the device and interacting with the microcontroller over USB, a ST **LIS2DH12** accelerometer to measure 3-axis motion and vibration, a TI **MAX4618** multiplexer to handle communication and mating to the AlphaSense frontend board, a Maxim **DS3231** RTC (from the same family as the DS1307 used in learnAir V1 but with temperature compensation and much more accurate timing), and

standard breakout headers for several I2C and SPI environmental sensing peripherals at both 5V and 3.3V. The main circuit is powered off of 3.3V to save on power, but 5V power handling is included. Power to 5V peripherals can be programmatically shut on and off by the microcontroller for power saving when sleeping high quiescent devices. Schematics for learnAir V2 can be found in Appendix B.

Two daughter boards were designed to connect with the main board learnAir board. This modular design gives more flexibility for housing the device, simple upgradability, and separates concerns when testing/verifying the circuitry.

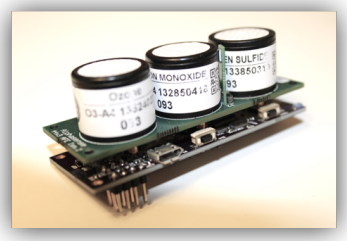


Figure 10: Second revision, Atmel based learnAir main board mated with the AlphaSense sensor frontend

The first of these daughter boards includes all sensors that need to be mounted on against the edge of the device, either in contact with the air or with direct access to sunlight. This small board includes a ROHM **BH1730FVC** I2C light sensor, a Vishay **VEML6070** I2C UV sensor, and a Sensirion **SHT25** temperature and humidity sensor. The SHT25 is similar to the SHT21 used in learnAir V1, but with slightly tighter tolerances.

The second daughter board is designed to hold three Omron **D6F-PH** differential pressure sensors, for 3-axis wind sensing. This is the same pressure sensor as was tested in the learnAir V1 device. Since the D6F-PH does not come with selectable I2C addresses, this daughter board also has an NXP PCA9545 I2C-bus switch to selective open the I2C bus between one of the three pressure sensors and the main learnAir board. This I2C bus switch is itself addressable and controllable over I2C, so no extra pins are required to connect it. Schematics for both daughter boards can be found in Appendix B.

Besides the connections to the three AlphaSense gas sensors controlled by the AlphaSense Analog Front End board, the main board is also equipped to connect to either a Sharp GP2Y1010AU0F (as tested in the learnAir V1 device) or the much more accurate $500 AlphaSense **OPC-N2**. The OPC-N2 takes advantage of Mie scattering algorithms and uses higher quality optics, but it is insensitive to particles under a few hundred nm and has a limited feature-set around flow filtering and control. Independent validation of the OPC-N2 has been mixed, and the device includes a built in fan (demanding a lot of power for a portable device), but it offers an enticing combination of cost and size given its sophistication.

A final physical design of the learnAir V2 device was modeled in SolidWorks, and fits in the palm of a user's hand (Figure 11). The

final cost of the OPC-N2 version is around $1k, while the Sharp version is closer to $600.

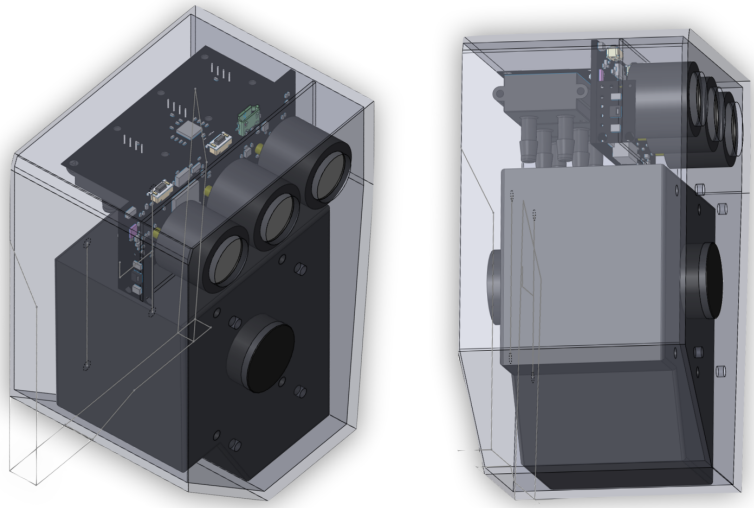


Figure 11: Final design of the portable system

Once the hardware was actually designed and built, firmware to control the device needed to be created. The Arduino environment and libraries were used to write code for learnAir V2. Since this is a demanding application, the standard pinouts used by typical Arduino boards needed to be redefined. LearnAir clocking and custom pin mapping definition files were added to the Arduino environment– now, similar to 'Leonardo' and 'Due' and other boards, the 'learnAir' board can be chosen and programmed as an option from the Arduino environment dropdown menu. Firmware and board definition code is available in Appendix B. The completed code and hardware was tested with a learnAir smartphone application and shown to work reliably.

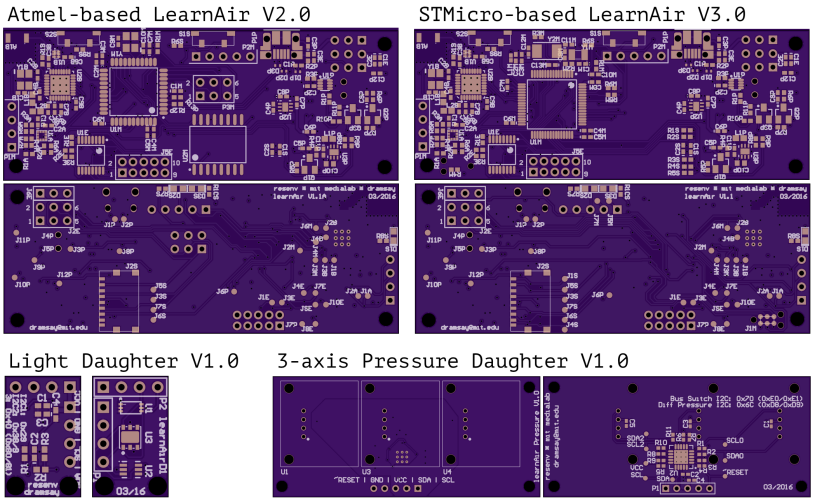


Figure 12: Layouts for revisions 2.0 and 3.0 of the learnAir board

LearnAir Version 3

The third version of learnAir hardware is very similar to the second in most ways. It supports the same daughter boards, peripherals, power circuitry, USB charging/communication, accelerometer, and bluetooth communication. The main difference is the choice of microprocessor– this time, the ST Micro STM32L152 was used. This processor is much more advanced than the Atmel chip built into version 2– it offers more a elegant programming interface, it has higher resolution ADCs (12-bit instead of 10-bit), it runs at much lower power and with many advanced power-handling features, it includes advanced features like an on-board RTC and a fully parallel SDIO interface, it has extensible and powerful code support, and it brings the entire board down in price.

While it is able to connect to the existing wind daughter board, this hardware includes an experimental differential pressure sensor (for wind measurement) on the main learnAir board itself. This is a cutting edge MEMS differential pressure sensor– the Sensirion **SDP31**– which is not yet publicly available. While it performs similarly the Omron D6F-PH, it is dramatically smaller– 8.5 x 5.5 x 4.4mm (600 mm^3) vs. the Omron's 26 x 18 x 22mm (31,000 mm^3). This is a huge reduction in size for the learnAir system.

The STM32L152's onboard RTC also saves in cost and layout space compared with version 2. The fully parallel SDIO interface is helpful for power savings– SD Card data can be written over SPI (with two

data lines) or over this 4-pin parallel bus. SD Card writes are power hungry, so it is useful for low-power operation to be able to minimize write time by maximizing the parallelization of data transfer. BLE transfer also requires bursts of power– it is valuable to be able to store data on-board and batch-send it to the phone or over USB, especially if the device's battery is low. Additionally, the extra SPI and I2C buses allow parallelization, which means we can cut our duty cycle down. This type of advanced power optimization is a goal of learnAir V3.

Power consumption for both boards is dominated by SD card writes, BLE, and high power peripherals. Rough power estimates of each system show the LearnAir V2 system requiring around 5 mW of power (attached to the Sharp sensor with a 30 second duty cycle), and the V3 system requiring around 3.5 mW. Since the OPC-N2 is a relatively high power device, and must be left on for at least one second to acquire a reading (unlike every other sensor in the system), if it is connected it completely dominates our power estimates. Both systems require an order of magnitude more power with the OPC-N2 included (35-40 mW). These are very rough estimates and a measured power characterization is required for meaningful claims– it is safe to say, however, that with a several hundred mAh LiPo battery we can expect a battery life of several days to several weeks, depending on the attached peripherals and duty cycling of the device.

The V3 version– while built and tested– still requires a few extra support libraries to be ported over before it is smart-phone ready. Firmware for the ST Family is much more complicated, and requires much more thorough understanding of documentation and setup to optimize it for truly low-power operation. The build process and configuration of the board, its programming, and basic measurement and output controls have all been successfully tested with this design. It represents a major step toward true production-quality. Schematics and code samples can be found in Appendix B.

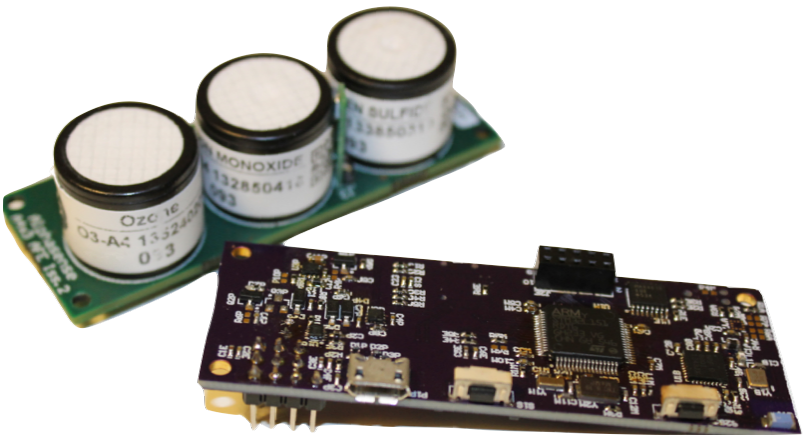


Figure 13: Third revision, STM32L152-based learnAir main board next to the AlphaSense sensors

Hardware Comparison and Analysis

MassDEP hardware uses robust, FEM certified techniques– thus it can be taken as a grounded reference over the time-scales it measures. The corresponding air quality sensors in the learnAir system are variable in quality, and their analysis is discussed in Chapter 7. Most of the remaining sensors included in the learnAir system are robust and well-characterized for their purpose.

To validate proper function, in this section we compare temperature and humidity readings from the learnAir device against corresponding weather API data retrieved from ForecastIO for the latitude and longitude of the device. We also analyze and compare our experimental differential pressure wind sensing design against the ground-truth MassDEP wind speed and wind direction.

ForecastIO data is hourly, and a 60 minute rolling average is used to interpolate the values to the minute timescale. LearnAir data is collected every minute.

Temperature and Humidity

Humidity was recorded in the box by the SmartCitizen Kit and compared against the ForecastIO reading. Figure 15 shows a comparison of each SmartCitizen reading with each ForecastIO value– ideally they would track closely, falling on the 1:1 line when plotted against each other. Instead we see a slight skew towards higher humidity in the box. It is reasonable to assume this is a real phenomena– temperature differential in the box may cause condensation and elevated humidity. In either case, the consistency between the measurements is quite good, and suggests trustworthiness.

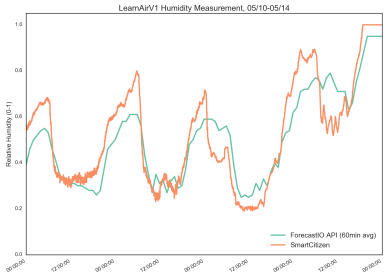


Figure 14: Humidity Comparison of SmartCitizen (orange) and ForecastIO (green) over 4 days

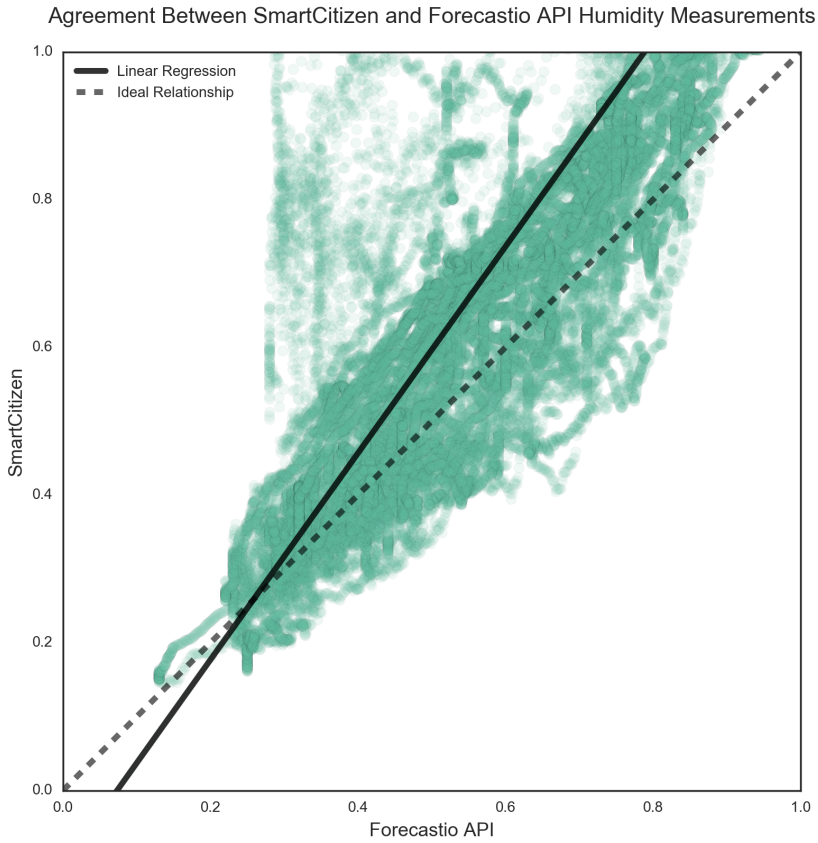


Figure 15: Humidity Comparison, SmartCitizen and ForecastIO

Temperature was recorded inside the box by both the AlphaSense temperature sensor and the SmartCitizen Kit. Figure 16 shows quantization error on the raw AlphaSense reading (teal)– this is mitigated by taking a 15-minute rolling average (orange). In Figure 17 we see a four day comparison of ForecastIO data with data taken in the box from the AlphaSense and SmartCitizen sensors. We see good agreement between in-the-box data, with some slight variation as temperatures exceed 25 degrees Celsius. There is also good agreement between the ForecastIO data and the in-the-box data when the sun is down. This is a real effect– the learnAir box was exposed to direct sunlight, and thus shows significant rises in temperature during the day compared with ambient conditions. Both temperatures (and their differential) are used to as features for our machine learning algorithm, with in-the-box temperatures informing our calibration process (which is appropriate as the air quality sensors are likewise in the box).

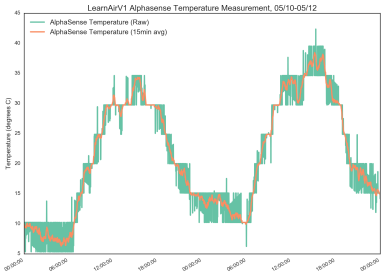


Figure 16: AlphaSense Raw Temperature Data (green) with 15-minute averaging (orange)

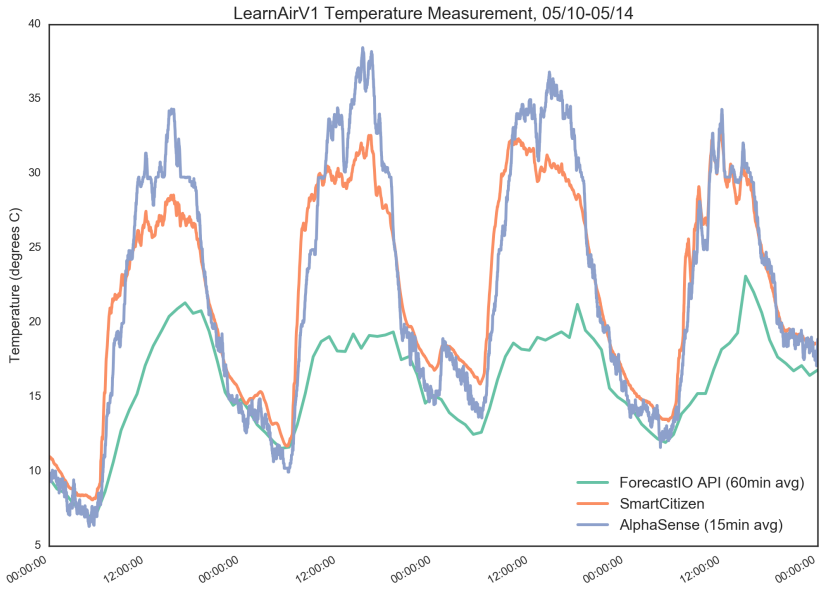


Figure 17: Temperature close-up

Wind

Using differences in pressure to sense wind direction is not a new concept– pitot tubes (frequently found on airplanes) are one such example. Recent work to create MEMs wind sensors using this modality has also started to appear. [80] It is not typical to find differential pressure sensing used in this way, however; particularly at this scale, and taking into account (or potentially leveraging) the geometry of a larger box.

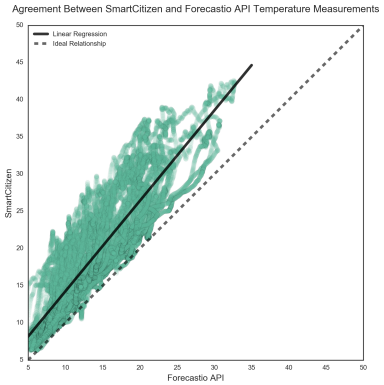


Figure 18: Temperature Comparison, SmartCitizen and ForecastIO

Thus, the included differential pressure sensor was a small, cheap, and experimental way to measure airflow and wind through the device. One of its two ports is connected to the side of the box with a tube, while the other is vented into the box– since the box has holes and is pressure equalized, any differences we measure can be attributed to wind on the outside box face. While the learnAir V1 sensor only had one differential pressure sensor in it (mounted on the same face as the other sensor inlets), the boards were designed to support three axis sensing, in order to get a truly 3-D understanding of wind speed and directivity and how that may affect air quality measurement. Doing so accurately requires polar pattern analysis, orthogonality in wind response vs. direction, and some interesting

device geometry and signal processing. This is an open research question in and of itself. In this first instance, only one axis (measuring speed and not direction) was captured on the primary face of the device.

As a first step to explore the feasibility of such a wind sensing system, the learnAir system was (1) characterized using a home-made laminar flow setup, and then (2) compared against trustworthy external wind speed and direction data from MassDEP.

The goal of the first test was to get a sense for learnAir's wind direction selectivity. Since the design is rectangular and the wind sensor is protected by a slotted design, we would expect air flow parallel with the slots to penetrate less than perpendicular flow. We'd also expect air coming directly at the sensing face of the device to penetrate much more than air flow coming at the back.

Figure 19: A picture of a simple laminar flow test setup for rough wind directivity characterization

To test this, PVC pipes filled with large straws was placed in front of a fan to approximate laminar flow conditions (Figure 19). Cardboard was placed around the laminar flow section to prevent spurious eddy currents. A handheld Extech 45158 Anemometer was used to validate a constant airflow of 2 m/s (a light breeze). The learnAir V1 device was then placed in the flow, and rotated at 30 degree intervals, with 10 wind measurements at each interval. The top half of figure 20 shows a normalized polar response with air flowing directly towards the slotted wind sensor device face at 0 degrees (and directly at the rear face of the device at 180). The bottom half shows the response of the device with wind flowing over top of the face at various angles– 0 degrees represents airflow parallel with the short dimension of the learnAir box, and 90 degrees is parallel with the long dimension.

As expected, the device responds with an interesting 3 dimensional pattern. It is very responsive to airflow coming at the face, and very insensitive to air flow coming from behind. It is very sensitive to airflow perpendicular to its slots, and completely insensitive to airflow that is parallel. This is a very useful feature to exploit for truly three dimensional wind sensing, and for controlling device airflow. Designs that include checkerboard slots or slots in both directions may be very impervious to wind. At the same time, with no active airflow through the device, these results suggest that the air that is being sampled is (1) more likely to be pushed into the cavity from specific directions, which may artificially affect its sensitivity to directional sources of pollution like a nearby road, and (2) we may expect some low-pass filtering effects relative to a sensor design that actively pulls air through the device, especially when the wind is blowing in a direction that has difficulty penetrating the slotted cover.

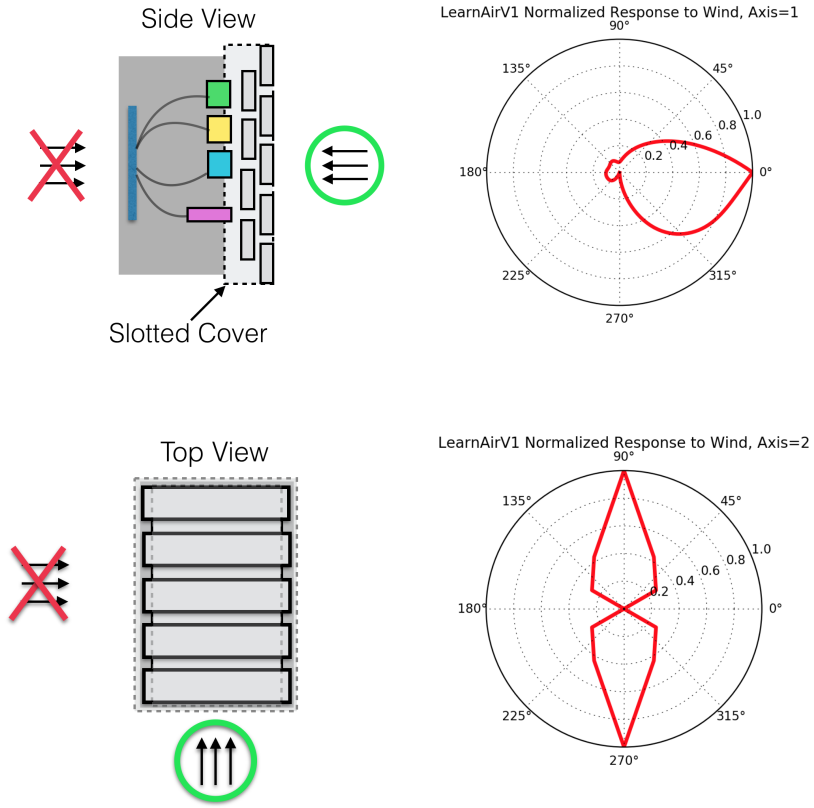


Figure 20: Wind Directivity Polar Patterns

The second test compares windspeed measured by the device with windspeed measured externally by the MassDEP sensor. We would expect, given our polar plots, that (1) the actual airflow we're sensing is different/shielded from the real wind, so there may be some differences in the measurement, and (2) our device is selective to certain wind directions, so it is important to analyze the relationship of errors in our readings compared with the MassDEP readings as a function of wind direction.

Figure 21 shows a comparison of measured windspeed data from our pressure sensor against the MassDEP data for one day (after preconditioning the signal and LMSE scaling it against the MassDEP reference). There is a clear correlation in overall trend, suggesting the pressure sensor is capturing meaningful information. Tight agreement of $\pm 5\%$ between readings is highlighted in green.

While the overall trend is there, there are some large discrepancies. Based on our polar plots, it seems worthwhile to look at the differ-

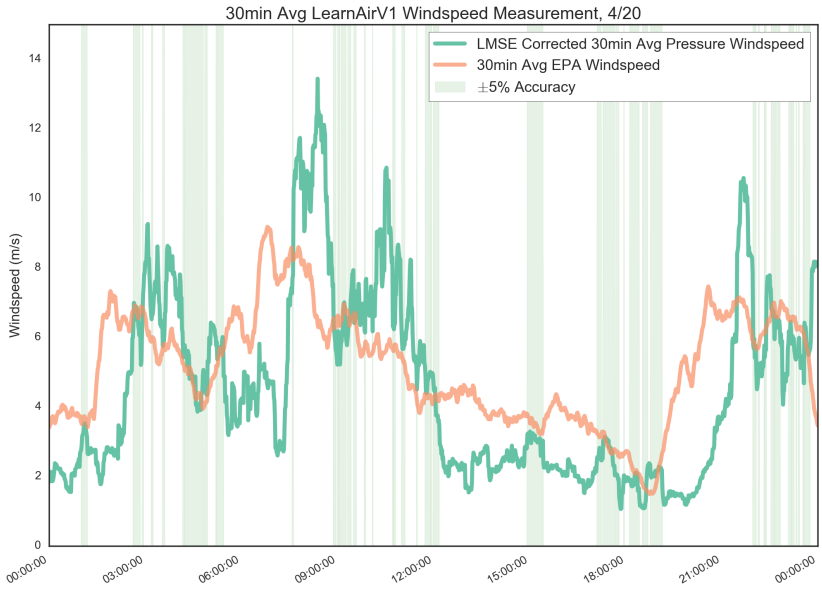


Figure 21: Wind Speed Measurement with 10% Accuracy, Zoomed

ences between our measurement and the MassDEP measurement as a function of wind direction, as shown in Figure 22. There are clear and interesting relationships between measurement inaccuracies and wind direction– there are large differences in the readings when the wind approaches from 60, 120, 210, 270 degrees, while 0, 90, 180, 260, and 320 degrees seem to match more closely. This does not corroborate expectations exactly, given our polar plots (0 and 180 degrees having low error since they allow wind to pass, and 90 and 270 degrees having high error since they reject airflow). These results suggest a more complicated relationship between direction and selectivity. More rigorous testing is required to accurately characterize the directionality of this system.

The overall trends support the fact that our pressure sensor is measuring airflow in a correct and useful way. Having a measure of airflow inside the slotted casing is good for tracking meaningful penetration of wind, regardless of direction. This work suggests that pressure sensors have great potential to provide a low cost, small option for accurate wind sensing. Future research is necessary to optimize sensor geometry, orthogonalize sensor axes, and characterize the effect of device geometry and turbulence on system linearity, in order to understand the wind sensing potential for this technology. For this project, the pressure signal provides insight regarding near-field air flow that can be used as a training feature for our machine learning model. More figures describing the wind data can be found

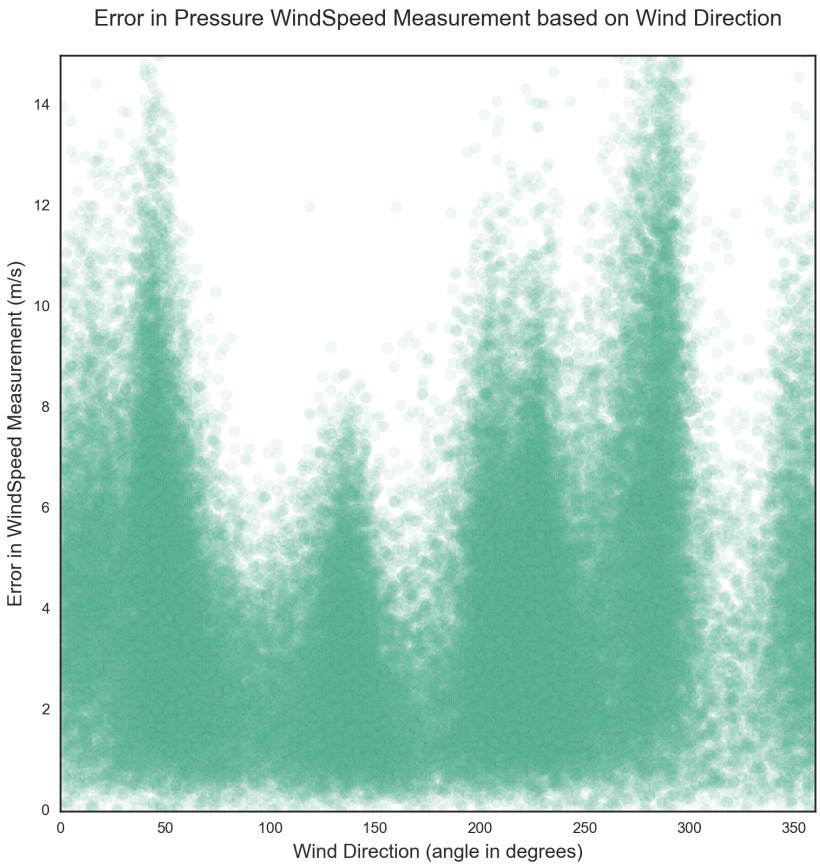


Figure 22: Discrepancy in Windspeed Measurement vs Wind Direction

in the Appendix.

6. *ChainAPI for Air Quality*

ChainAPI [39, 40] is a hypermedia framework for the 'Web of Things'. It provides a minimal layer of design principles on top of the HAL/JSON specification, to make data sharing and resource addressing simple. It is extensible– allowing anyone to define their own ontologies and connect their own data storage solutions in a distributed fashion– and attempts to only rigorously define a thin, hyperlink layer that allows information to be easily discoverable and easily digestible by any user or service.

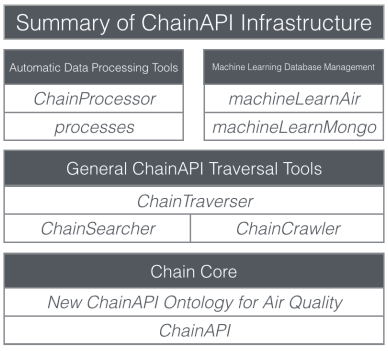


Figure 23: Summary of New ChainAPI Infrastructure

ChainAPI lends itself to the kinds of problems facing the air quality community. It allows anyone, with any type of device, to store their data however they see fit, and still easily connect to a broader ecosystem. It allows researchers to browse through the entire ecosystem like they would the internet. It incentivizes contribution with a broader data ecosystem and interesting tools, while simultaneously lowering the barrier to entry as much as possible.

In the Responsive Environments group at the MIT Media Lab, ChainAPI already forms the backbone of a large ecological sensor network installation called TidMarsh. [41] It connects hundreds of sensors to a browsable, easy to use backend. It provides a simple real-time data stream that developers have connected to for building virtual environments, data visualizations, audio compositions, and other novel tools for data interaction. Besides addressing the core concerns of distributed, scalable, simple data-sharing, ChainAPI also stands as the backbone of several future-looking human interactions and interventions to help individuals live with, understand, internalize, and mediate large datasets.

The TidMarsh ChainAPI environment forms the basis of the ChainAPI Air Quality protocol. To make it useful for air quality, the first step is to define an ontology that addresses the needs and concerns of the

air quality community. This ontology is significantly larger and more complex than the current ChainAPI ontology, catering specifically to concerns of stationary and mobile air quality sensing. This includes shared resources that define sensor types and device types and relevant metadata, as well as Organizational and Fixed Site information so that larger groups like the EPA have an example resource that maps well to their current ontology.

While several interesting tools have been created on top of the core ChainAPI for interacting with live streams of known resources, there are no tools to automatically explore and interact with a data ecosystem that is dynamic and growing, or an ecosystem that has reached critical mass (i.e. one update stream becomes infeasible as a way of monitoring all resources). These are features we would expect from a true, distributed 'Web of Things' solution, and are of particular concern for an air quality installation designed to support deployments ranging from the EPA to small citizen groups. These features also lay the groundwork for scalable learning algorithms, that can search the network for nearby higher quality sensors and utilize their data.

In this thesis, we define a new ontology for ChainAPI to adapt it to the air quality space. After creating a development environment that takes advantage of this ontology, we created several tools to enable scalable interaction and advanced learning on this dataset. These tools– chainCrawler, chainSearcher, chainTraverser, and chainProcessor– form a backbone of extensible, powerful options for resource discovery, automatic and transparent cross-organization dataset creation and processing, as well as scalable, advanced machine learning techniques. It encourages programmatic data processing that can scale and is easily trackable. It also encourages a separation of concerns– so the best-in-class raw data collection and the best-in-class preprocessing algorithms can both exist transparently and be applied broadly, instead of siloed researchers batch processing and scrubbing data with opaque techniques before any data is shared.

At it's most advanced, these tools provide a simple way to find and compare co-located sensors of similar type but disparate quality, easily access the conditions under which those measurements occurred, and apply any user-defined algorithm. This is all scaffolded in such a way that the underlying algorithm or model can simply and automatically update as more data is added to the network.

A New Ontology for Air Quality

The first major step in adapting ChainAPI for air quality networks is to define a new ontology. The basic outline is as follows:

> *Organizations have Deployments. Deployments have Fixed Sites and Mobile Devices. Both Fixed Sites and Mobile Devices have an extensible API datastore for weather conditions, etc. (corresponding to their location) associated with them. Fixed Sites have (non-Mobile) Devices. Devices each are associated with a particular Device Type (make and model), and each Device has a collection of Sensors (individual data streams). Sensors are associated with a particular Sensor Type, and have a collection of SensorData.*

There are other resources and details, which are outlined in the full documentation below. Some data (like SensorData, CalibrationData, and APIData) have an associated storage resource that contains general, important information about the collection of data (like what it's measuring, which API it is calling, etc). LocationData, on the other hand, requires no metadata, and thus does not require a storage resource. 'Type' resources are useful for quickly identifying resources of the same type, and centralizing information about how to handle that type.

One interesting example of the utility of centralized resource types is that manufacturers could potentially oversee their device and sensor types– updating metadata and associated calibration algorithms. A new user to the ChainAPI system could then simply link their sensor resource to the correct type, and an automated crawler could pull the most recent, manufacturer-specified calibrations and apply them automatically to the new user's data. It could check the conditions under which a measurement was made automatically, and warn the user if the sensor is out of normal operating ranges. Additionally, the manufacturer could store sensor service information in their sensor type, and have an automated crawler that looks for sensors of that type, checks their serial numbers, and notifies the contact person associated with a resource if their sensor is in need of recalibration.

In general, the principle behind data storage in ChainAPI is to create many 'virtual sensors' to represent one real one. We encourage users to store raw data in Chain, and after processing it, post the processed data to a parallel 'virtual sensor' that has a name that indicates it comes from the same physical sensor, but has new units or is a new metric.

A few minutiae are important for interoperability. Timestamps are stored using ISO8601 standards (timezone aware UTC)– all python code and scripts will accept (and only accept) one of the major timezone-specified string formats. Timed/averaged measurements are stored with a 'start time' and a 'duration'.

Organization

An installation of ChainAPI maintained and curated by an Organization.

name(string) – the name of the organization.
url(string) – the website URL associated with the organization.
ch:deployments (related resource) – a collection of deployments associated with the organization.
ch:contacts (related resource) – a collection of contacts associated with the organization.

Deployment

A particular project owned by an organization, usually with several to hundreds of mobile sensors and/or fixed sites.

name(string) – the name of the deployment.
geoLocation(elevation, latitude, longitude) – a location to associate with the deployment; usually, the city where the deployment is based.
ch:organization (related resource) – the parent organization.
ch:sites (related resource) – a collection of fixed sites associated with the deployment.
ch:devices (related resource) – a collection of mobile devices associated with the deployment.
ch:contacts (related resource) – a collection of contacts in charge of the deployment.

Fixed Site

An immovable location where several devices are co-located.

name(string) – the name/identifier of the site.
url(string) – a website URL reference with extra documentation about the site.
geoLocation(elevation, latitude, longitude) – the fixed coordinates/elevation of the site.
ch:deployment (related resource) – the parent deployment.
ch:devices (related resource) – a collection of devices located at the fixed site.
ch:calibration_datastores (related resource) – a collection of calibration datastores associated with the site.
ch:api_datastores (related resource) – a collection of API datastores associated with the site location.
ch:contacts (related resource) – a collection of contacts in charge of site maintenance.

Device

A manufactured object that houses a collection of sensors. If it is mobile, its proper parent is a deployment. If it is

stationary, its proper parent is a fixed site.

unique_name(string) – a unique identifier for the device.
serial_no(string) – the manufacturer's unique identifier for the device.
deploy_date(ISO8601 timestamp) – the date the device was put in the field.
manufacture_date(ISO8601 timestamp) – the date the device was manufactured.
description(string) – extra, descriptive data pertaining to this device.
ch:device_type (related resource) – general device information for this make/model device.
ch:site (related resource) – the parent site, if a fixed device.
ch:deployment (related resource) – the parent deployment, if a mobile device.
ch:locationDataHistory (related resource) – a collection of timestamped location data, if a mobile device.
ch:api_datastores (related resource) – a collection of API datastores associated with the device's timestamped locationData.
ch:contacts (related resource) – a collection of contacts in charge of the device.

Device Type

A collection of useful data pertaining to several devices of the same make and model.

manufacturer(string) – the name of the device manufacturer.
model(string) – the name/number of the device model.
revision(string) – the hardware revision of the device.
datasheet_url(string) – the website URL associated with the most current datasheet.
description(string) – extra, descriptive data pertaining to this device type.
ch:devices (related resource) – a collection of all devices of this type.

Sensor

An object that captures a single channel of timestamped data. There maybe multiple sensors in a device.

metric(string) – a label for the measured quantity (i.e. 'O3', 'CO', 'NO2').
unit(string) – the units of the measurement (i.e. 'ppb', 'ug/m3')
ch:sensor_type (related resource) – general sensor information for this make/model sensor.
dataType(string) – the datatype stored by the sensor (float).
value(float) – the most recent reading from the sensor.
updated(ISO8601 timestamp) – the timestamp of the most recent reading from the sensor.
ch:dataHistory (related resource) – the collection of timestamped data from this sensor.
ch:device (related resource) – the parent device.

Sensor Type

A collection of useful data pertaining to several sensors of the same make and model.

manufacturer(string) – the name of the sensor manufacturer.
model(string) – the name/number of the sensor model.
revision(string) – the hardware revision of the sensor.
datasheet_url(string) – the website URL associated with the most current datasheet.

description(string) – extra, descriptive data pertaining to this sensor type.
retail_cost(float) – rough estimate of sensor cost, for future 'value' comparisons.
learn_priority(int) – an indication of sensor trustworthiness. This can be used so lower ranking sensors will learn from higher ranking ones automatically.
service_interval_days(float) – number of days between recommended servicing.
sensor_topology(string) – a description of device operating principles (i.e. 'BAM', 'electrochemical')
ch:sensors (related resource) – a collection of all sensors of this type.

Sensor Data

The raw data associated with a sensor.

dataType(string) – the datatype of the raw data (float).
totalCount(int) – total number of saved datapoints, or the number returned on this page if the dataset is large.
data (list) – a collection of data objects, each with a 'value' (float) and a 'timestamp' (ISO8601 timestamp)

API DataStore

An object that captures a single channel of external API data. There maybe multiple API Datastores associated with a mobile device or a fixed site.

metric(string) – a label for the measured quantity (i.e. 'temp', 'humidity).
unit(string) – the units of the measurement (i.e. 'Celsius', 'percent')
metadata(string) – an extra text field for relevant metadata.
ch:api_type (related resource) – general information about the external API.
dataType(string) – the datatype stored in this API datastore (float).
value(float) – the most recent value from the API.
updated(ISO8601 timestamp) – the timestamp of the most recent API call.
ch:dataHistory (related resource) – the collection of data from this API.
ch:site (related resource) – the parent site if associated with a site.
ch:device (related resource) – the parent device if associated with a mobile device.

API Type

A collection of useful data pertaining to a given external API.

api_name(string) – the name of the API.
api_base_address(string) – the base API access URL.
description(string) – extra, descriptive data pertaining to this API.
ch:devices (related resource) – a collection of all mobile devices that use this API.
ch:sites (related resource) – a collection of all fixed sites that use this API.

API Data

The raw data associated with an API Datastore.

dataType(string) – the datatype of the raw data (float).

totalCount(int) – total number of saved datapoints, or the number returned on this page if the dataset is large.
data (list) – a collection of data objects, each with a 'value' (float), a 'timestamp' (ISO8601 timestamp), the 'api_call' used to retrieve the data (string), the 'api_access_time' (ISO8601 timestamp) when the call was initiated, and the 'duration_sec' (int) that the API data is useful for (starting from 'timestamp').

Calibration DataStore

An object that captures a single channel of calibration data. There maybe multiple calibration datastores associated with a mobile device or a fixed site.

metric(string) – a label for the measured quantity (i.e. 'sensitivity', 'voltage_offset').
unit(string) – the units of the measurement (i.e. 'ppb/nA', 'mV')
metadata(string) – an extra text field for relevant metadata.
dataType(string) – the datatype stored in this calibration datastore (float).
value(float) – the most recent calibration value.
updated(ISO8601 timestamp) – the timestamp of the most recent calibration.
ch:dataHistory (related resource) – the collection of data from calibration datastore.
ch:site (related resource) – the parent site if associated with a site.
ch:sensor (related resource) – the parent sensor if associated with a mobile device.

Calibration Data

The raw data associated with a Calibration Datastore.

dataType(string) – the datatype of the raw data (float).
totalCount(int) – total number of saved datapoints, or the number returned on this page if the dataset is large.
data (list) – a collection of data objects, each with a 'value' (float), a 'timestamp' (ISO8601 timestamp), a 'description' (string), and a 'contact' (related resource).

Location Data

The raw data that forms a collection of timestamped location information for tracking a mobile device.

latitude(float) – a latitude GPS coordinate.
longitude(float) – a longitude GPS coordinate.
elevation(float) – elevation in meters.
timestamp(ISO8601 timestamp) – the timestamp associated with this location reading.
ch:device (related resource) – the parent device.

Contact

A person that is part of an organization, oversees/calibrates a deployment or site, or owns a device.

first_name(string) – a contact's first name.
last_name(string) – a contact's last name.
phone(string) – a contact's phone number.

```
    email(string) – a contact's email address.
    ch:organization (related resource) – a contact's organization.
    ch:deployments (related resource) – a collection of deployments overseen by the contact.
    ch:devices (related resource) – a collection of devices owned by the contact.
    ch:sites (related resource) – a collection of sites overseen by the contact.
    ch:calibration_data (related resource) – a collection of calibration logs measured by
        the contact.
```

Traversing ChainAPI

chainCrawler - a web-crawler for ChainAPI

Now that we've created an ontology for air quality, it's important to have the tools to interact with the data as new devices are added. ChainCrawler is a tool for crawling through ChainAPI resource links and discovering new resources. It works like a traditional web-crawler.

ChainCrawler is highly optimized for speed and scale, using Google's CityHash to track the most recently visited resources so the crawler doesn't loop or backtrack. It has the additional feature of tracking hash collisions as required, and can accept any power of 2 size hash table.

ChainCrawler accepts an entry point URI, and picks a random, unexplored link to traverse from that resource. If it reaches a dead end or has already visited all of a resource's links, it moves back through its recent history (of URIs in history are definable) to look for unexplored resources. If it runs out of history, it returns to the entry-point resource. At this point, if every entry-point resource path has been visited, the cache is cleared and the process is started over.

ChainCrawler will return the URI(s) of chain resources based on search criteria. It can filter on resource_type (i.e. 'Site' or 'Device'), resource_title (i.e. 'Site 1- Roxbury' or 'Device 2'), any arbitrary object attribute, or any combination of the above.

ChainCrawler can be used in several modes. It can be run in a blocking manner, and simply return the URI of the first resource it finds. It can be run as a separate thread, and pass URIs to another thread using python's 'Queue' library. It can also be run in ZMQ push mode,

in which case all URIs are pushed out over a ZMQ socket using push/pull (preferred method). In these threaded cases, chainCrawler will not stop crawling until forced. It will not return duplicate resources for over a given, user-definable, refractory period (which can be set to infinite).

---

chainCrawler.ChainCrawler(entry_point, cache_table_mask_length, track_search_depth, found_set_persistence, crawl_delay, filter_keywords)

Initialize a ChainCrawler Instance.

entry_point (= 'http://learnair.media.mit.edu:8000') is the URI of the resource to start crawling.
cache_table_mask_length (= 8) is the exponent used to define the hash mask for the hash table. (hash table size = 2^cache_table_mask_length)
search_depth (= 5) is the number of URIs we store in history, in case we exhaust all links and have to back up.
found_set_persistence (= 720) is the time, in minutes, that a crawler will remember a resource it has already seen, and will not re-push it to the user
crawl_delay (= 1000) is the time, in ms, in between calls to the server to access chain resources.
filter_keywords (= ['next','previous']) is an array of link relationships we want to ignore while crawling. For our data, 'next' and 'previous' are required.

chainCrawler.ChainCrawler.find(namespace, resource_type, plural_resource_type, resource_title, resource_extra)

Blocking crawl that will exit/return the URI of the first matching resource.

namespace (= "") is the base URI that defines the ontological relationships. Prepended to resource_types.
resource_type (= None) is an optional resource type search criteria that must match a given resource for it to be returned.
plural_resource_type (= None) is a search criteria that should correspond the resource_type field. Plural types are automatically generated by looking at the singular resource_type and adding 's' and 'es', but for words that have strange pluralization, it is important to give the correct plural form.
resource_title (= None) is an optional resource title search criteria that must match a given resource for it to be returned.
resource_extra (= None) is an optional dictionary of attribute:value pairs that must match a given resource for it to be returned.

chainCrawler.ChainCrawler.crawl_thread(q, namespace, resource_type, plural_resource_type, resource_title, resource_extra)

Similar to find, but this function spins up a background thread crawler that will push URIs of the matching resources onto the queue 'q'.

q (= None) is the Queue object that URIs will be pushed to for other python threads to

```
            access.


chainCrawler.ChainCrawler.crawl_zmq(socket, namespace, resource_type, plural_resource_type,
    resource_title, resource_extra)

        Similar to find, but this function spins up a background thread crawler that will push URIs of the matching resources
        over a PUSH/PULL ZMQ socket.

        socket ( = 'tcp://127.0.0.1:5557') is the ZMQ PUSH/PULL socket that URIs will be pushed
            to for other programs to access.
```

```
1  crawler = ChainCrawler('http://learnair.media.mit.edu:8000/',
       ↪ found_set_persistence=2, crawl_delay=500)
2
3  #--- Blocking Find ---#
4      x = crawler.find(namespace='http://learnair.media.mit.edu:8000/rels/',
           ↪  resource_type='sensor', resource_extra={'sensor_type':'
           ↪ Alphasense03-A4'})
5      print x
6
7  #--- Threaded Queue ---#
8  testQueue = Queue.Queue()
9  crawler.crawl_thread(q=testQueue, namespace='http://learnair.media.mit.edu
       ↪ :8000/rels/', resource_type='Device', resource_title='test004')
10
11 #caution: this main loop doesn't end
12 while True:
13     uri = testQueue.get()
14     print uri
15
16 time.sleep(5)
17
18 #--- ZMQ Socket Push on TCP://127.0.0.1:5557 ---#
19 crawler.crawl_zmq(namespace='http://learnair.media.mit.edu:8000/rels/',
       ↪ resource_title='Device #1')
```

chainSearch - a breadth first search tool for ChainAPI

ChainCrawler allows us to randomly crawl through the entire chain infrastructure looking for a specified resource or resource type. While this is powerful, it is also important to have a generalized tool that allows us to search for closely associated resources– for instance,

locating the sensors that are part of a device if we only know the device's URI. Crawling would be the wrong strategy in this case.

Instead, we want to examine all resources linked directly from the device to see if any match our query, and then examine the child relationships of those resources, and so on. Perhaps we *only* want the resource if it is a direct child. In these cases, chainSearch- a breadth first search tool that will search within a specified number of link relationships away from the entry resource- is the correct tool for the job. Combined with chainCrawler, we now have tools to crawl and find any/all resources in ChainAPI, and then intelligently traverse local relationships within the ChainAPI ecosystem.

---

chainCrawler.ChainSearch(entry_point, crawl_delay, filter_keywords)

Initialize a ChainSearch Instance.

entry_point (= 'http://learnair.media.mit.edu:8000') is the URI of the resource to start crawling.
crawl_delay (= 1000) is the time, in ms, in between calls to the server to access chain resources.
filter_keywords (= ['next','previous']) is an array of link relationships we want to ignore while crawling. For our data, 'next' and 'previous' are required.

chainCrawler.ChainSearch.find_degrees_all(namespace, resource_type, plural_resource_type, resource_title, degrees)

Blocking, exhaustive breadth first search of all resource 'degrees' degrees away from the entry point. Returns a list of all URIs within 'degrees' of links from the current resource that match the search criteria. Returns an empty list if no resources match criteria.

namespace (= "") is the base URI that defines the ontological relationships. Prepended to resource_types.
resource_type (= None) is an optional resource type search criteria that must match a given resource for it to be returned.
plural_resource_type (= None) is a search criteria that should correspond the resource_type field. Plural types are automatically generated by looking at the singular resource_type and adding 's' and 'es', but for words that have strange pluralization, it is important to give the correct plural form.
resource_title (= None) is an optional resource title search criteria that must match a given resource for it to be returned.
degrees (= 1) is the number of degree relationships to exhaustively search from the entry point for a matching resource.

chainCrawler.ChainSearch.find_first(namespace, resource_type, plural_resource_type, resource_title, degrees)

Same as find_degrees_all, but this blocking breadth first search will immediately exit and return a resource upon

finding the first match. If it exhaustively searches within degrees, it returns an empty list.

degrees (= 3) is the number of degree relationships to exhaustively search from the entry point for a matching resource.

chainCrawler.ChainSearch.find_create_link(namespace, resource_type, plural_resource_type, degrees)

Exhaustive breadth first search that returns the first matching create-form URI for a resource of type resource_type.

degrees (= 1) is the number of degree relationships to exhaustively search from the creation link of type 'resource_type'.

chainCrawler.ChainSearch.reset_entrypoint(new_entrypoint)

Helper function to reset 'entry point' of a ChainSearch instance for easy re-use.

new_entrypoint (= 'http://learnair.media.mit.edu:8000') is a string matching the URI of the new starting point resource.

```
1  searcher = chainSearch.ChainSearch('http://learnair.media.mit.edu:8000/
       ↪ devices/10')
2
3  #-- Find Creation Link to Make a Sensor on this Device --#
4  resource_uri = searcher.find_create_link(namespace='http://learnair.media.
       ↪ mit.edu:8000/rels/', resource_type='sensor' )
5  print resource_uri
6
7  #-- Change Starting Point for Search --#
8  searcher.reset_entrypoint('http://learnair.media.mit.edu:8000/devices/?
       ↪ site_id=1')
9
10 #-- Find First Device with Name 'Device #3' within 4 Link Steps --#
11 resource_uri = searcher.find_first(resource_title='Device #3', degrees=4)
12 print resource_uri
13
14 #-- Find All Devices within 2 Link Steps --#
15 list_of_resource_uris = searcher.find_degrees_all(resource_type='Device',
       ↪ degrees=2)
16 print list_of_resource_uris
```

chainTraverser - a stateful spider for ChainAPI

Normally we think of web spiders as simple crawlers. chainTraverser is a Spider that comes with extra functionality. ChainTraverser always 'sits' on top of an associated ChainAPI resource. It takes advantage of chainCrawler and chainSearch in a natural way– from its current resource, it can (1) crawl randomly to another resource of a certain type, (2) search and traverse basic link relationships, (3) search and traverse complicated, pre-defined link paths, and (4) move forward and backwards relative to where it has been.

Besides moving through ChainAPI from one resource to another, chainTraverser provides many tools to interact with the ChainAPI resource it is associated with. From the current resource, chainTraverser can add new child resources, or even add cascading child resources along a pre-defined link path. It can also pull and push data to and from its resource.

For all air quality applications, this is the primary tool used to GET/-POST data to ChainAPI, as well as to handle traversal tasks from the trivial (i.e. finding the temperature sensor belonging to a device) to the extensive (i.e. creating a multi-sensor, multi-device site given a target organization name and deployment).

---

chainTraversal.ChainTraversal(crawl_delay, entry_point, namespace)

Initialize a ChainTraversal Instance.

crawl_delay (= 1000) is the time, in ms, in between calls to the server to access chain resources.
entry_point (= 'http://learnair.media.mit.edu:8000') is the URI of the resource to start crawling.
namespace (= 'http://learnair.media.mit.edu:8000/rels/') is the base URI that defines the ontological relationships. Prepended to resource_types.

chainTraversal.ChainTraversal.print_state()

prints the current state- the associated resource, the resource type, and the traversal history.

chainTraversal.ChainTraversal.back()

moves the traverser back to the previous node.

chainTraversal.ChainTraversal.forward()

moves the traverser forward to the next node, if we've moved back in the history.

chainTraversal.ChainTraversal.find_a_resource(resource, name)

Blocking crawl to a resource of type 'resource', specifically one titled 'name' if name is passed. Upon completion, chainTraversal is associated with this resource.

resource is the resource type the traverser will crawl for and then update its state to reflect.
name (= None) is name or title of the resource query we are crawling to match. If left as None, the first resource of the correct type will be considered a match.

chainTraversal.ChainTraversal.add_a_resource(resource_type, post_data, plural_resource_type)

This adds a resource of resource_type with attributes in post_data, one link from the current traversal node. This is safe to call if a resource already exists- it will return False if a matching resource is found. Otherwise, it will create the resource and return the server response.

resource_type the type of resource to be created (i.e. 'Device', 'Sensor')
post_data the required data to create the corresponding resource_type
plural_resource_type (= None) is a search criteria that should correspond the resource_type field. Plural types are automatically generated by looking at the singular resource_type and adding 's' and 'es', but for words that have strange pluralization, it is important to give the correct plural form.

chainTraversal.ChainTraversal.move_to_resource(resource_type, name, plural_resource_type)

This is designed to move to a neighboring resource with a link relationship to the current traversal node. If no matching resource is found with the shallow breadth first search, the traverser will not move and the function will return False (successful traversal returns True).

resource_type is the resource type the traverser will search for and then update its state to reflect.
name (= None) is name or title of the resource query we are searching to match.
plural_resource_type (= None) is a search criteria that should correspond the resource_type field. Plural types are automatically generated by looking at the singular resource_type and adding 's' and 'es', but for words that have strange pluralization, it is important to give the correct plural form.

chainTraversal.ChainTraversal.add_and_move_to_resource(resource_type, post_data, plural_resource_type)

This is a combination of the previous two functions, to create and move to a resource. It is safe to call if a resource already exists- it will not overwrite it, it will simply move to it.

chainTraversal.ChainTraversal.find_and_move_path_exists(path_list)

This expects a list of dicts that will guide us through Chain API. It will crawl to find the first resource, and then move through the following items one link relationship at a time, if they exist, until settling on the last node. If the path is incorrectly specified, it will move as far down the path_list as possible.

path_list a list of dicts specifying types and names of existing resources to traverse. i.e.,[{'type':'organization', 'name':'testOrg Name'}, {'type':'deployment', 'name':'learnairNet'}, {'type':'device', 'name':'device1'}]

chainTraversal.ChainTraversal.find_and_move_path_create(path_list)

This expects a list of dicts that will guide us through Chain API. It will crawl to find the first resource, and then move through the following items one link relationship at a time, creating resources if they don't exist, until settling on the last node.

path_list a list of dicts specifying types of resources and post_data to traverse and/or create. i.e., [{'type':'organization', 'name':'testOrg Name'}, {'type':'deployment', 'post_data':{'name':'learnairNet'}}, {'type':'device', 'post_data':{'name':'device1'}}]

chainTraversal.ChainTraversal.add_data(post_data, resource_type)

This adds data to a sensor, calibration datastore, api datastore, etc. if the traverser is located at that node. It will post data without checking for existing data, and may result in duplicate copies. If this is undesirable, use ChainTraversal.safe_add_data().

resource_type (='dataHistory') the type of data resource to be created. For normal sensors, the appropriate resource_type is the default 'dataHistory'.
post_data any array of properly formatted data values to post

chainTraversal.ChainTraversal.safe_add_data(post_data, resource_type, max_empty_steps)

This adds data to a sensor, calibration datastore, api datastore, etc. if the traverser is located at that node. It will first pull data at the node where it should be posting. If data exists at matching timestamp to post_data, these data won't be uploaded/duplicated.

resource_type (='dataHistory') the type of data resource to be created. For normal sensors, the appropriate resource_type is the default 'dataHistory'.
post_data any array of properly formated data values to post
max_empty_steps (=10) for pulling the comparison data from the sensor. This is the number of empty pages in a row we must see before we assume we've reached the end of the data stored in Chain (there is no explicit indication of earliest/latest measurements, it must be assumed by traversal).

chainTraversal.ChainTraversal.get_all_data(max_empty_steps, resource_type)

This pulls all data from a sensor, calibration datastore, api datastore, etc. if the traverser is located at that node. It will return a timestamp sorted list of dicts with the relevant data.

max_empty_steps (= 10) the number of empty pages in a row we must see before we assume we've reached the end of the data stored in Chain (there is no explicit indication of earliest/latest measurements, it must be assumed by traversal).
resource_type (= 'dataHistory') the type of data resource to be pulled. For normal sensors, the appropriate resource_type is the default 'dataHistory'.

```
traveler = ChainTraversal()
traveler.print_state()

#-- Standard Traversal --#
traveler.find_an_organization('ResEnv Test Organization #1')
traveler.move_to_resource('deployment', 'Test Deployment #2')
traveler.back() #back to organization
traveler.forward() #forward to deployment
traveler.move_to_resource('device', 'testdevice001') #move to device
traveler.add_and_move_to_resource('sensor', {'metric':'COT','sensor_type':
    ↪ 'alphasenseCOT','unit':'ppb'})

#-- Add Data to Current Sensor --#
traveler.print_state()
traveler.safe_add_data(
        [{'value':64.1,'timestamp':'2016-05-18 20:09:00+0000'},
         {'value':65.0,'timestamp':'2016-05-18 20:11:00+0000'},
         {'value':62.2,'timestamp':'2016-05-18 20:12:00+0000'},
         {'value':64.3,'timestamp':'2016-05-18 20:13:00+0000'},
         {'value':63.9,'timestamp':'2016-05-18 20:14:00+0000'},
         {'value':63.7,'timestamp':'2016-05-18 20:15:00+0000'} ])

#-- Path List Examples --#
find_and_move_path_exists([{'type':'organization', 'name':'testOrg Name'},
    ↪  {'type':'deployment', 'name':'learnairNet'}, {'type':'device', '
    ↪ name':'device1'])

find_and_move_path_create([{'type':'organization', 'name':'testOrg Name'},
    ↪  {'type':'deployment', 'post_data':{'name':'learnairNet'}}, {'type':
    ↪ 'device', 'post_data':{'name':'device1'}}])
```

ChainAPI Tools for Scalable, Automatic Data Analysis

chainExcelPush - easily add spreadsheet data to ChainAPI

One important part of the ChainAPI infrastructure to drive its adoption are basic tools for data manipulation and uploading. chainExcelPush is a simple, extensible tool that uses chainTraversal to make uploading excel files and csv files quick and painless.

chainExcelPush provides one prototype file and requires one command to use properly. The prototype file must be edited to (1) provide the general path through ChainAPI to the device where you would like to push the data, and (2) provide a mapping between each excel column label and the device's sensor in ChainAPI where it should post. Once this prototype function has been filled out, running the chainExcelPush script will prompt the user to select the path to their excel files, and the rest will happen automatically.

This is an example of a prototype function for the LearnAir V1 device. By modifying this 'smart_upload' function, it is easy to automate any excel upload to chainAPI:

```
def smart_upload(upload_array):
    #look at keys, figure out where these values should be stored in chain
    #call upload and actually upload values

    def switch(x):
        return {
            'humidity ( % raw)':{
                'device':{
                    'unique_name':'learnAirFixedV1',
                    'device_type':'learnAirFixedV1'},
                'sensor': {
                    'sensor_type':'SHT21',
                    'metric':'humidity_raw',
                    'unit':'raw'},
                                    },

            'light ( lx)':{
                'device':{
                    'unique_name':'learnAirFixedV1',
                    'device_type':'learnAirFixedV1'},
                'sensor': {
```

```
                    'sensor_type':'BH1730FVC',
                    'metric':'light',
                    'unit':'lux'},
                                    },

            'nitrogen␣dioxide␣(␣kohm)': {
                'device':{
                    'unique_name':'learnAirFixedV1',
                    'device_type':'learnAirFixedV1'},
                'sensor': {
                    'sensor_type':'MICS4514',
                    'metric':'NO2_raw',
                    'unit':'kOhm'},
                                    },

            'alphas1_aux': {
                'device':{
                    'unique_name':'learnAirFixedV1',
                    'device_type':'learnAirFixedV1'},
                'sensor': {
                    'sensor_type':'Alphasense03-A4',
                    'metric':'03_raw_aux',
                    'unit':'raw'},
                                    },


                    ...<many other mappings>...


            'sharpdust': {
                'device':{
                    'unique_name':'learnAirFixedV1',
                    'device_type':'learnAirFixedV1'},
                'sensor': {
                    'sensor_type':'GP2Y1010AU0F',
                    'metric':'PM25_raw',
                    'unit':'raw'},
                                    }
        }.get(x.lower(), None)

    for key in upload_array.keys():
        learnair_data_upload(

                [{'type':'organization', 'name':'MIT␣Media␣Lab'},
```

```
                {'type':'deployment', 'post_data':{'name':'LearnAirTestDev
                    ↪ '}},
                {'type':'site', 'post_data':{'name':'RoxburyEPA'}}],

                switch(key),
                upload_array[key])
```

In this example, the key in the switch array describes the excel or csv column label, and the associated object defines the device and sensor where data should be uploaded. The non-variable part of the path (describing the organization, deployment, and site) is given below, in a list form. After describing this mapping, any files can be uploaded by simply running the script from the command line, which will initiate a search prompt for folders/paths to relevant excel files:

```
>> ./chainExcelPush.py
```

chainProcessor - scalable, automatic learning for ChainAPI ecosystems

The previous tools for ChainAPI allow us to easily navigate ChainAPI and perform basic data manipulations with resources and data. These tools come together in a more advanced tool-kit with chainProcessor– a set of classes designed to scaffold and promote scalable ChainAPI algorithms. ChainProcessor crawls through chain, pulls out any data associated with a given type of sensor, and provides an easy interface for scientists and data analysts to: (1) write simple raw data processing functions, (2) write more sophisticated calibration algorithms, and/or (3) write highly sophisticated, auto-updating, predictive machine learning algorithms. In all of these cases, 'virtual sensors' representing processed data, calibrated data, or predicted data are posted back into ChainAPI alongside the original sensor stream.

The cornerstones of chainProcessor are the chainProcessor routine and the 'processes' folder, which has a prototype process example file. The processes folder contains files that must be labeled with the name of a corresponding ChainAPI Sensor Type (i.e. 'AlphaSenseO3-A4.py'). The chainProcessor routine automatically pulls the process names from the processes folder and crawls ChainAPI for the sensor types that match a defined process. When a resource is found, it queries the process file to see if additional local data is required for the data processing step, and then mediates dataflow to and from the

process. A prototypical process file for an SHT21 temperature sensor is shown below:

```
#SHT21Temperture.py

#every process must have required_aux_data and process_data routines
#every process should have its own dispatcher and processing functions
#called from the dispatcher

import numpy as np
from .. import machineLearnDatastore

def dispatcher(metric, unit):
    #this tells which extra data are required and which functions to use
    #to process a given metric/unit combination for this sensor type

    return {
        'temperature_raw': { 'raw': {
            'function': raw_to_temp
            }},
        'temperature': { 'celsius':{
            'extra_data': ['humidity_corrected', 'light_corrected'],
            'function': temp_to_learned_temp
            }}

            }.get(metric, None)[unit]


#all functions should return [sensor_type, metric, unit, data_to_post]

def raw_to_temp(data):
    #processes raw SHT21 readings to accurate temperature using datasheet
    #equation posts to a 'SHT21Temperature' sensor measuring 'temperature'
    #in 'celsius' units on the same device as the 'raw' SHT21 temp. data

    for data_element in data:
        data_element['value'] =  -50.0 + 175.72 * (data_element['value'] *
            ↪  10 / (2**(16) ) )

    return('SHT21Temperature', 'temperature', 'celsius', data)


def temp_to_learned_temp(data):
    #passes us 'humidity_corrected' and 'light_corrected' data from the
```

```
41      #same device if it exists, to use for a more advanced calibration
42      #that accounts for humidity/light effects
43
44      for data_element, humidity in zip(data['main'], data['
            ↪ humidity_corrected']):
45          data_element['value'] =  -50.0 + 175.72 * (data_element['value'] *
                ↪  10 / (2**(16) - 0.05 * humidity['value'])
46
47      return('SHT21Temperature', 'temperature', 'celsius_learned', data['
            ↪ main'])
48
49
50  #---- DO NOT EDIT THESE FUNCTIONS ----#
51
52  def required_aux_data(metric, unit):
53      #logic for extra data required by this module: for instance, if we
54      #have an alphasense NO2-A4 sensor, if we have a raw working electrode
55      #data we also need raw aux electrode data and perhaps raw temp data to
56      #make sense of the reading and create a virtual sensor.  This returns
57      #a list of required secondary data for the sensor_type of this file,
58      #and the metric/unit of that type, so that the main process routine
59      #can traverse, find that extra data, and pass it back to process_data
60      try:
61          return dispatcher(metric, unit)['extra_data']
62      except:
63          return None
64
65
66  def process_data(data, metric, unit):
67      #call logic - depending on metric/unit, call subprocess
68      #return processed data and metric/unit to post
69      try:
70          return dispatcher(metric, unit)['function'](data)
71      except:
72          return None
73
74  #---- DO NOT EDIT THESE FUNCTIONS ----#
```

In this function, the user edits the dispatcher and writes data processing functions. The dispatcher defines two key relationships for a given sensor type, metric, and unit combination– it defines auxiliary data, and it defines the name of the function in the process file that will accept, modify, and return a processed version of that

data. The auxiliary data is assumed to be local data on the device or at the site– for instance, local temperature or humidity readings, or auxiliary electrode readings, that are required to properly calibrate a sensor stream. For more advanced algorithms and machine learning techniques that require extra information and extra state beyond what is available from the sensor and its neighbors, support functions discussed below can be integrated into these processes to enable them to learn and update their model over time. These scripts are user-definable and can be run on any chainAPI instance.

Thus, the main process routine has four main jobs. First, it crawls and finds sensor data that matches the defined process. Then it looks at the process dispatcher, and determines which extra local data the process needs. It pulls all of this data from the resource, and stamps every datapoint with latitude and longitude information based on its timestamp, and the available location data (for mobile devices) or the GPS coordinates (for fixed devices and sites). This data (indexed on timestamp, latitude, and longitude) is used to call the relevant process function. The main process then waits for a return list of post_data for a virtual sensor– the sensor_type, metric, unit, and an array of data. This return list is posted to ChainAPI at the parent device of the original sensor.

This structure is already interesting and useful- instead of pre-processing data and pushing it after a set calibration or after running opaque, complicated data processing, this model allows (1) easy updating of calibration and data processing algorithms as they get better or more refined, (2) more transparent sharing of raw data, processed data, and processing routines, and (3) open the opportunity for manufacturers to track their own hardware, flag anomalies, qualify sensors as they age, and own/improve/update the routines used for calibrating their sensors.

While this is interesting, the end goal is to enable sophisticated machine learning with this technique. To that end, a support module called machineLearnDatastore comes into play.

For the purposes of machine learning, we divide our sensors up into 'conditions' and 'measures'. This is an arbitrary distinction, but generally speaking 'conditions' are reliable, trusted, simple measurements and API calls that will be used to predict the accuracy of more complex and less reliable air quality 'measures'. It is possible, using this technique, to easily add any 'measure' to a 'conditions' array when desired.

machineLearnMongo is a helper class that can be used in any process. It creates a MongoDB database, with one consolidated collection for all 'conditions' (things like temperature and humidity), and one independent collection for each 'measure' (things like O_3 readings from AlphaSense O_3-A4 sensors). MachineLearnMongo takes care of saving all of the data whenever a new update comes in, and consolidating conditions into one table. All data in machineLearnMongo are indexed by timestamp, latitude, and longitude.

The most important part of machineLearnMongo is the 'get_values_in_range' and 'create_ml_array' functions. The first of these allows searching any collection for values within a given time and location window. The closest values to the ideal time and/or location are returned if multiple values fall within the window. The second function automatically pulls all data from a 'measure' collection, and finds all the associated 'conditions' (and additional specified measures) that were measured within the specified time/location window of each measure datapoint. This one simple command will examine all measurements that have been crawled by chainProcessor, match all data that was coincident, and return a full array of features that can be used to predict and train a machine learning model for the specified 'measures' array.

At the moment, the user must define which other sensor types a given sensor should learn from. However, fields are included in this ontology to automate that process– giving a rank to each sensor, so that lower quality sensors can automatically learn from any higher quality reference. This field can be entered manually (by an independent and trustworthy testing organization), or programatically by an algorithm that tracks sensor quality and revises rankings accordingly.

Additionally, the get_values_in_range function returns the actual distance and delay between two measurements so it may be accounted for when training on the comparison data. As the dataset grows, it is possible to automatically and rigorously converge on the useful range of distance and time offsets (simply by using this information as another feature in the machine learning feature vector).

---

machineLearnDatastore.machineLearnMongo(db)

Initialize a MongoDB database handler instance. This is optimized for machine learning, and everything is indexed by timestamp, latitude, and longitude.

db (= 'learnair') is the name of the Mongo database

machineLearnDatastore.machineLearnMongo.create_indexed_collection(collection_name)

Initialize a MongoDB collection in our database, indexed by timestamp, latitude, and longitude. This is used to initialize collections of 'measures'.

collection_name is the name of the new Mongo collection that will be created.

machineLearnDatastore.machineLearnMongo.create_conditions_collection(collection_name)

Initialize a MongoDB collection in our database, indexed by timestamp, latitude, and longitude. This is used to initialize the database's collection of 'conditions'. As new conditions are added, only

machineLearnDatastore.machineLearnMongo.add_data_to_collection(collection_name, data)

Add data to one of the Mongo 'measures' collections.

collection_name is the name of the Mongo collection we'd like to add data to.
data is the data to be added. Data should be formed as [{'timestamp':x, 'lat':y, 'lon':z, 'fieldtoadd':xyz}, {'timestamp':x, 'lat':y,'lon':z, 'fieldtoadd':xyz}].

machineLearnDatastore.machineLearnMongo.add_conditions(data)

Add data to one the 'conditions' collections.

data is the data to be added. Data should be formed as [{'timestamp':x, 'lat':y, 'lon':z, 'fieldtoadd':xyz}, {'timestamp':x, 'lat':y,'lon':z, 'fieldtoadd':xyz}].

machineLearnDatastore.machineLearnMongo.print_collection(collection_name)

Print collection_name collection to assess data and structure.

collection_name is the name of the Mongo collection we'd like to print.

machineLearnDatastore.machineLearnMongo.print_conditions()

Print 'conditions' collection to assess data and structure.

machineLearnDatastore.machineLearnMongo.get_collection_data(collection_name, query)

Access data from the 'collection_name' collection.

collection_name is the name of the Mongo collection we'd like to query.
query (= {}) is the query to use on the collection. If left blank, all documents in the collection will be returned.

machineLearnDatastore.machineLearnMongo.get_conditions_data(query)

Access data from the 'conditions' collection.

query (= {}) is the query to use on the collection. If left blank, all documents in the collection will be returned.

machineLearnDatastore.machineLearnMongo.return_ml_array(collection_name, conditions, measure, extra_conditions, update_conditions_first, time_range, lat_lon_range, loc_then_time, return_diffs)

Creates and returns a machine learning useful array. In summary, it takes the values from the collection_name, finds conditions in the conditions array that are taken at similar timestamps/latitudes/ longitudes, and combines them into an array where conditions can be used to predict measures.
This array is of the form:

```
[{'conditions':{'keya':val, 'keyb':val}, 'measures':{'keya':val, 'keyb':val}},
 {'conditions':{'keya':val, 'keyb':val}, 'measures':{'keya':val, 'keyb':val}},
 {'conditions':{'keya':val, 'keyb':val}, 'measures':{'keya':val, 'keyb':val}}]
```

Specifically it adds the fields in 'measure' from the 'collection_name' collection (and takes all of them if none are specified), and pulls 'conditions' from the conditions collection (pulling all fields if none are specified) that match the timestamp, latitude, and longitude of the measures. When there is a match- i.e., when we have conditions that line up with a measure- a new row in the returned machine learn array is formed. Instead of having to index an exact lat/lon/timestamp match (which is nearly impossible), this function takes a time_range and a lat_long_range where it considers measurements coincident. If more than one condition value of the same type qualify as coincident, loc_then_time can be used to set the more important feature (i.e. whether the closer value in time or the closer value in proximity takes priority). If multiple conditions entries are 'coincident' but have different fields, the full union of fields will be returned. Overlapping fields within these will prioritize the 'closer' condition.

collection_name is the name of the Mongo 'measure' collection for which we'd like to apply machine learning.

conditions {= None) are the fields stored in the conditions array we'd like to add to our machine learning array. If unspecified (None), all fields are used.

measure {= None) are the fields stored in the collection_name array we'd like to add to our machine learning array. If unspecified (None), all fields are used.

extra_conditions {= None) is a dictionary of extra collections/fields that we'd like to add to our conditions array (for instance, if we want to add a cheap NO2 sensor 'measure' to our 'conditions' array to predict the accuracy of a cross-sensitive O3 sensor). This dictionary should be of the form {collection_name:[field1, field2], collection_name:[field1, field2]}.

update_conditions_first {= True) will run through all indices in our conditions array (indexed by timestamp/lat/lon) and add API call data to each where it is missing, before constructing this array.

time_range {= 30) the time range, in seconds, for which two measurements are considered 'coincident'.

lat_lon_range {= 1) the range, in degrees, for which two latitude or longitude measurements are considered 'coincident'.

loc_then_time {= True) if two of the same value are coincident with the measure in collection_name, prioritize the one closer in distance instead the one closer in

```
            time if True.
        return_diffs {= True) add metadata for the difference in lat/lon/time between the '
            measure' collection and the 'conditions' into the conditions array (i.e. 'time_diff
            ', 'lat_diff', 'lon_diff', 'distance')


machineLearnDatastore.machineLearnMongo.get_values_in_range(collection_name, timestamp, lat, lon
    , time_range, lat_lon_range, loc_then_time, return_diffs)

        Return one document from collection_name that is the closest fit to timestamp/lat/lon in the ranges
        specified (within 30 seconds, 1 degree of lat and lon by default). If there are no documents, return None.
        time_range, lat_lon_range, loc_then_time, and return_diffs all operate the same with the same
        defaults as return_ml_array above.
```

Instead of interacting directly with the machineLearnMongo class, the machineLearnAir class was created as an intermediary. It provides a scaffolding for adaptive, scalable machine learning algorithms.

MachineLearnAir is designed to manage a machineLearnMongo class, as well as typical machine learning algorithms that require a computationally intense 'training' phase. When data is passed to a machineLearnAir instance, it is written to a machineLearnMongo database. The class tracks when a training step was last run, and if enough new samples (more than update_model_with_x_new_entries) have been added since the last update, the training step is triggered to run on all of the data stored in the machineLearnMongo instance managed by the class. This training step then updates a Mongo representation of the machine learning model.

Regardless of whether the training step is called or not, data passed to the machineLearnAir instance will be passed to the main machine learning algorithm. This step recalls the model trained in the previous step and apply it to the input data, returning processed data. Both the model and the last_updated state are saved, so stopping the process will not remove the most recent machine learning model.

Any user can add a new machine learning algorithm to the class simply by writing two functions– an 'algorithm_training' function, and an 'algorithm' function. When passing data to the instance, simply giving the argument algorithm='algorithm' will use the newly added code.

These functions are scaffolded in the example below– machine learning input data is pulled from the local machineLearnMongo collec-

tion, and the model (once trained in the training phase) should be stored with a model.post(<state>) command. The main algorithm function can access that model using the model.get() command, and should return a list of processed data.

In short, this function manages all of the hard parts of machine learning. Simply 'run' it on all data from a given sensor, and it will apply the latest machine learning model of that sensor to the data and return it. Whenever it accumulates a large enough batch of new data, it triggers the '_training' step that retrains the model using all available machine learning data. It automatically recognizes its specified algorithms, so it is easy to add a new algorithm by simply writing the 'algorithm' function and the 'algorithm_training' function as shown in the prototype file below. This makes it easily extensible for all types of new algorithms and techniques.

```
from machineLearnDatastore import *

learner = machineLearnAir(collection_name="AlphaSenseNO2" ,
    ↪ update_model_with_x_new_entries=500)

post_data = learner.run(input_data, algorithm='svm')
```

```
machineLearnAir.py

<...>

def svm_train(self, model):

        input_data = self.mongo.get_ml_array() #uses current collection

        <...training a ML model on input_data and store it in model_state
            ↪ ...>

        model.post(model_state)


def svm(self, data, model):

        ml_model_state = model.get()

        <...apply the model to the data...>

        return processed_data
```

Summary

ChainAPI represents a unique and powerful solution to data sharing and data interaction for sensor networks. It facilitates a scalable, distributed ecosystem while promoting easy interoperability and low barriers to entry.

In this thesis, we've adapted the ChainAPI ecosystem to address the needs of air quality community. This new ontology takes into consideration existing practices, as well as future needs of major research groups, citizen scientists, consumer product designers, and air quality equipment manufacturers. The development implementation was built to run and interface with our learnAir hardware and stream data to/from a smartphone application.

On top of the ChainAPI solution, we've also added all of the new infrastructure explained in detail above. We wrote these new tools *(chainSearcher, chainCrawler, chainTraverser, chainProcessor, machineLearnAir, and machineLearnMongo)* that look towards a scalable, dynamic future for ChainAPI, as well as advanced functionality for scraping ChainAPI, running machine learning algorithms that constantly update as new data is added to the ecosystem, and reposting processed data to the system. This scaffolding– a hypermedia layer to connect the air quality data ecosystem supporting crawlers that run through the ecosystem processing data, updating their data processing models using that data, and posting their processed data back to the ecosystem to be further assessed– is a novel topology for database design with interesting separation of concerns and transparency.

Besides the big picture ramifications of such a system, it allows us to accomplish our immediate goal– (1) build a sensor that can post its data to chain, (2) easily scrape the database for nearby, coincident measurements of other sensors and ambient conditions, (3) run and update a machine learning script to predict the accuracy of that sensor's measurement, and (4) subscribe to a live feed of that processed data so we can display and update the live prediction of that sensor's reliability based on all of the latest ChainAPI data.

7. Data Analysis and Machine Learning

In the last two chapters we outlined the portable hardware we built to measure air quality and push it up to ChainAPI. We discussed our new back-end infrastructure to support air quality data, as well as the tools that were created to allow automatic and extensible machine learning algorithms. The final element of the system– and the key element of this project– is the evaluation of predictive machine learning algorithms in the air quality space. In practice, proximity to quality sensors may be transient and at varying ranges and angles. Here we lay a preliminary foundation for such a system using optimal conditions.

For this test, we co-located six types of low-cost air quality sensor next to EPA reference-level equipment for two months. In most cases we collected minute-resolution data. The testing occurred over the course of spring (with high variation in weather as we transitioned from the end of the cold/snowy winter to the summer).

This experimental design gives us the ability to take the difference between our low-cost sensor signals and the EPA reference to generate an error signal. Based on measurements from other sensors on the device in concert with data from external weather APIs, we can apply supervised learning techniques that attempt to predict the magnitude of the error for every reading.

While there are many potential algorithms we could apply to this problem, for this experiment we simplified our error readings to a binary feature– indicating whether the cheap sensor was in error or not, based on an appropriate tolerance around the ground truth value. This tolerance is empirically chosen based on the 'noise level' of the device to be the smallest tolerance possible that still provides predictable results (i.e., the tolerance is the smallest it can be while remaining dominated by systematic failure modes instead of the

intrinsic measurement imprecision). A logistic regression model– commonly used for engineering failure prediction– was selected to predict whether the sensor was in error, as well as assign a probability to this prediction. This results could be used for many applications, for example the creation of a pollution map that optimally leverages data from all pollution sensors everywhere.

Test Conditions and Data Collection Summary

The learnAir V1 sensor was first installed on April 6th, 2016 and taken down on April 14th. This preliminary test resulted in serious sensor corrosion and unuseable data.

The useful section of the co-location test ran from April 15th to June 13th (59 days). During that time, several tests were performed. In total we validated 8 different sensors representing 6 distinct sensor types over periods ranging from 21 to 59 days. Our co-located test sets range from 1,431 samples of hourly data up to 85,739 samples of minute-resolved data.

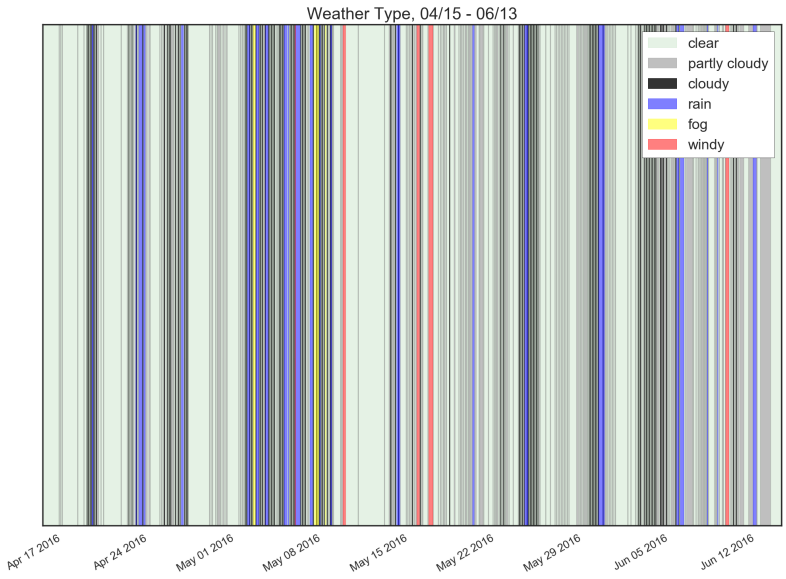


Figure 24: Weather during Test Period.

From April 15th through May 23rd (38 days with one 40 minute service break), an older set of AlphaSense O_3 and CO electrochemical gas sensors were characterized. This set was calibrated in Dec 2013 (2 years, 4 months old calibration at the start of the test). 55,589 samples of minute-resolution data was collected to characterize these sensors.

From May 23rd to June 13th (21 days), a brand new set of AlphaSense O_3+NO_2, NO_2, and CO electrochemical gas sensors were characterized (5 month old calibration at the start of the test). 30,150 samples were collected to characterize these sensors.

The two tests of an older and newer set of the same type of Alphasense sensors (O_3 and CO) can provide interesting insights– by comparing the data between the two tests we may be able to draw insights into age- and use-related differences. By combining the two sets after calibration, we have a long set of data spanning several months to test predictive techniques with. Assuming the underlying failure modes are the same for sensors of the same make/model, this combined set should be useful at providing insight into the underlying device mechanics.

From April 15th through June 6th (52 days, with two 40 minute service outages), a new SmartCitizen sensor– with its NO_2 and CO sensors– was installed and running at the MassDEP site. This sensor gave 85,739 minute-resolved samples.

Finally, from April 15th through June 13th (59 days), our Sharp Particulate Sensor collected samples that we resolved to 1 hour intervals, to match the MassDEP BAM reference. 1,431 samples were collected from this technique.

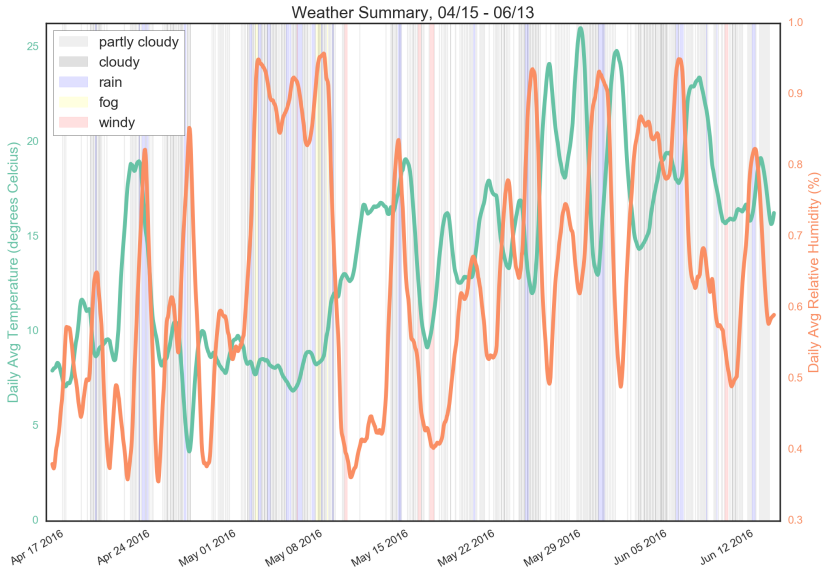


Figure 25: Temperature and Humidity during Test Period.

The co-location tests started with snow on the ground and ended with summer heat. External temperatures varied from below 5 to

above 30 degrees Celcius, and the relative humidity ranged from 20-100%. The weather during the period was nicely varied, with a few foggy days, a few windy days, and a particularly rainy week in early May. See Figures 24 and 25 for details, and Appendix D for more information about trends in ambient pressure, dew, light level, precipitation amount, and cloud cover over the course of the testing period.

Overview of Data Pre-Processing and ML Strategy

All readings taken by the co-located sensor were measured in 30 second intervals, and timestamped using the on-board Real Time Clock before being saved to an SD card. Since this was all done offline, no corrections for the timestamps could be applied.

In general, we found a 0.15-0.25% drift in the RTC timestamps. This results in 60-200 samples of error every three weeks or so. (The recorded duration and the actual duration of the measurements was off by less than 2 minutes over several weeks– the large number of missing samples comes from a slightly longer delay between measurements than 30 seconds. In our tests, this delay worked out to actually be between 30.05 and 30.08 seconds.) To appropriately align these drifted, 30 second values with our minute-resolved reference values, we applied two stages of correction. To begin, we corrected the timestamps to reflect their actual time (assuming a constant delay between each, and aligning their start and end times with the precise time the measurements started and ended as measured by a network connected smartphone). We then linearly interpolated between these corrected timestamp values to find on-the-minute values for every minute.

For the hour long samples of BAM data, we had to consolidate our minute data down to hourly values. To match the process of the BAM sensor (which collects samples for the first 50 minutes of every hour, and measures for the last ten), we wrote a script to average every value in our machine learning arrays over the course of the first 50 minutes of every hour, and throw away the last 10.

This is only the first step of pre-processing the data, however. For our air quality signals, a calibration step is necessary before comparing the data and characterizing the sensor. For the SmartCitizen CO and NO2 sensors, as well as for the Sharp particle sensor, their output

is an uncalibrated mV value. In the case of the Sharp sensor, the reading is inverted from the actual particle level (as it is a measure of light that makes it through the sensor without scattering). These sensors were calibrated using a Least Mean Squared Error (LMSE) approach to optimize a simple scale factor or a scale factor and an offset. The minimization function was run (1) on all of the data, (2) with the outliers (> 1 stdev) removed, (3) on long-term averages of the data, and (3) to only optimize values that were within some tolerance (throwing out sections that appeared to show disagreement and instead favoring a tighter fit on aligned data). The most realistic, quality fit was chosen from amongst these options as the 'calibrated' reference.

For the AlphaSense sensors, calibration was even more complicated. These sensors come with a calibration sheet giving appropriate values. The formula for most of their sensors is:

$$ppb = \frac{(we - we_{zero}) - (n * ae - ae_{zero})}{sensitivity}$$

where we is the working electrode measurement in mV, ae is the auxiliary electrode measurement in mV, we_{zero} is the working electrode offset value in mV, ae_{zero} is the auxiliary electrode offset value in mV, n is a temperature dependent and sensor chemistry dependent scale factor, and the sensitivity is the mV/ppb gain factor of the instrumentation amplifier on the conditioning board. For the cross-sensitive O3+NO2 reading, we use the calibrated NO2 values and subtract the resulting mV offset given the calibrated NO2 sensitivty:

$$ppb = \frac{(we - we_{zero}) - (n * ae - ae_{zero}) - \frac{no2_{ppb}}{x\_sensitivity\_to\_no2}}{sensitivity}$$

While we tried the provided calibration data– as well as simple LMSE scaling– we found the best agreement came from LMSE minimization using a bounded search of we_{zero}, ae_{zero}, and $sensitivity$. In these cases, the seed values were the provided calibration terms from AlphaSense. This type of calibration gave very strong results compared to other methods.

With calibrated data, an appropriate tolerance was chosen for each sensor value (typically $\pm$2-15% of the full range, though a larger tolerance was chosen for particulate, based on empirical observation of the smallest useful tolerance. Smaller tolerances that still provide predictable machine learning outcomes suggest better sensor physics). Each reading was then classified as 'correct' if falling within that tolerance of the MassDEP reference measurement, or 'incorrect' if falling outside of it.

This data is now ready for machine learning using the logistic regres-

sion discussed in the introduction to this chapter. We performed all of this analysis using python's scikit-learn machine learning toolbox– however, some experimentation was done with the java-based (GUI driven) 'Weka' toolkit (using a python ARFF file conversion library), as well as initial exploration with google's new tensor-flow library (which has logistic regression support, but which really shines for its deep learning ability and large dataset handling). These additions tools, and additional techniques, will be explored in more depth in the near future.

The general outline of the machine learning process we applied to the data is as follows: (1) load in the feature values (approximately 150 of them) to predict our binary error classification, (2) impute (or fill in) missing values, (3) split our data into training and test sets used 5-fold cross-validation, (4) run a grid search over logistic regression parameter-space to find the best regularization coefficient and penalty terms, (5) train our new 'best model' using the five training sets, and (6) verify the results on the five test sets. Importantly, two types of cross-validation are used– a shuffled type and a chunked type. In one case, data from the entire two month period is randomly selected to constitute training and test sets; in the other, the first several weeks are used to predict the last, the last several are used to predict the first, etc. The difference in these results gives us important insight into algorithm robustness and the effect of seasonal variation on our predictions.

Additionally, we use randomized decision trees to rank the importance of our features, as well as seven other feature reduction techniques (correlation, linear regression, random forest, lasso, RFE, ridge, and stability). A reduced set of the top 15 features is then used to retrain our original Logistic Regression, and the results are compared. The strength of agreement between feature reduction techniques can suggest meaningful predictive relationships, and the relative strength of the classifier with this reduced feature set can also corroborate strong causality for the top features.

There are two main metrics we use to evaluate our system performance. The most obvious metric is to compare the right answer ('is the sensor actually in error?') with the predicted one ('do we think the sensor is in error?') and display our results in a 2x2 confusion matrix. We can easily calculate the error rate of our algorithm from this matrix.

Logistic Regression offers a probability along with its prediction,

however, so to fully evaluate the strength of our results we must take these probabilities into account. Our second evaluation metric– and the accepted standard for this type of evaluation– is a Receiver Operating Characteristic (ROC) curve. This curve plots the true positive rate (the number of correct predictions that a measurement is in error, normalized by the total number of errored air quality readings) against the false positive rate (the number of incorrect predictions that a measurement is in error normalized by the total number of accurate air quality readings). We can compute a point on this graph by choosing an arbitrary threshold for our probability, and classifying every measurement as a predicting an error in our reading or not based on whether the probability that it is falls above or below this threshold value. If we set our probability threshold at 50%, we find the point corresponding to our original confusion matrix.

When we plot the points for every threshold value (from 0-100%), we generate an ROC curve. These curves start at (0,0) and end at (1,1) on our plot. Random guessing will form a line between the points at a 45 degree angle. Perfect accuracy with 100% confidence will form a right triangle– jumping immediately to a value of (0,1) on our graph before continuing horizontally over and meeting up with the upper right corner. Real, meaningful predictions will likely fall somewhere in between. The area under the ROC curve (AUC-ROC) is normalized to a value between 0 and 1, and frequently used to characterize the quality of predictions generated by our logistic regression in a more comprehensive way than a simple confusion matrix. Generally speaking, values above 0.8 suggest our model has good-to-excellent predictive power as the number grows closer to one, and values above 0.7 represent reasonable predictions. Values below this mark are marginal, with anything close to 0.5 suggesting total failure.

Once we've found the AUC-ROC for our optimized logistic regression, we compare the results for the 5-fold validation sets (one having been 'shuffled' or randomized, and the other having been 'chunked'). The shuffled version assumes no time dependent phenomena– using randomly chosen samples throughout the entire test to predict random other samples interspersed throughout the test. The chunked version is a more realistic model– using data we've already gathered from one period of time to predict future data.

We must be careful with the shuffled version– errors frequently occur together in time (a sensor will misread for an hour or two in a row, giving a few hundred errors at once, likely because of underlying

phenomena such as high relative humidity). Monotonically increasing functions (like temperature) could serve as a proxy for time, and the algorithm might take advantage of this co-occurrence to 'predict' our error. This is a classic example of overfit, and by simply looking at the underlying feature and error distributions one can ensure that the model hasn't led to incorrect conclusions. This could lead to artificially strong results in the shuffled case. If we see a strong predictive relationship for one of these variables in the shuffled case, it is important to make sure that the variable is not simply monotonic and co-occurring with one large window of consecutive poor readings.

After verifying the quality of our shuffled results, we can compare the shuffled and blocked versions of the algorithm. If the shuffled version still does substantially better than the block-wise version, it suggests that we haven't trained on enough data to capture season-agnostic predictive information. However, when the two agree, it forms a powerful indication that (1) we've captured enough data to train across seasons, and (2) we have hit upon real and useful phenomena.

Machine Learning Features

In most of these applications, around 150 features were used to train our machine learning algorithms. Features were either measured from the learnAir system or harvested from the Forecast.IO weather API. A few additional features (the EPA reference black carbon, wind speed, and wind direction measurements) were all included as training features. Raw sensor signals are included as well as their calibrated versions. Many of these signals were further manipulated or processed to give more derived features.

The measured set of features represents data collected from on-board sensors– temperature, humidity, noise, light, vibration, pressure/airflow, and other air pollution sensors. We also include the signal from the sensor whose quality we're trying to predict as well (if it reads in certain ranges or slews at a certain rate, we may be able to assume that the sensor is out of its useful range).

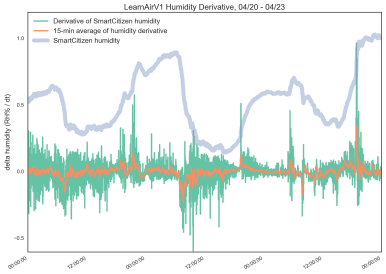


Figure 26: Humidity Derivative Feature Creation

For most of the main features, we created derivative features to predict errors resulting from rapid changes in environmental conditions (anecdotally, we know this to be true for electrochemical sensors). We also looked at intelligently chosen averages over time– ones that

minimized quantization errors, smoothed data to match the EPA reference, or whose averaging gave evidence of longer term trends that might also be important in analyzing sensor state and measurement quality.

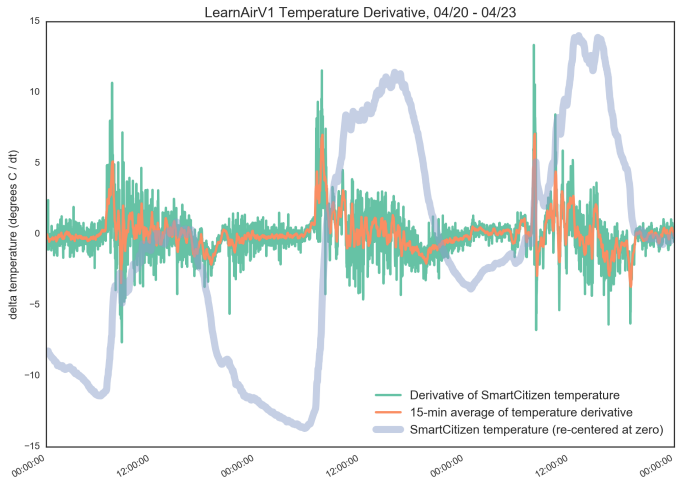


Figure 27: Temperature Derivative Feature Creation

Several API's were evaluated for use in this project, and Forecast.IO stood out as a well-reputed option. They use machine learning to combine many weather forecast APIs into one highly accurate dataset. The Forecast.IO data comes in hourly intervals, so a running 60 minute average was used to interpolate the values to minute resolution (most of the measured fields, like temperature, are relatively slow-moving). For class-based indicators (for instance, the 'weather-summary' field indicating 'cloudy', 'windy', 'foggy', 'rainy', etc) we converted the API value into binary fields that matched each class.

We created features such as 'temperature differential' and 'humidity differential' – a constructed feature that corresponds to the difference in measurement between the ambient conditions (as measured by Forecast.IO) and the conditions in the box (as measured by the SmartCitizen Kit). While these features are linear combinations of other features (and thus won't improve our model's predictive power), they serve an intuitive purpose, and may help reduce the feature set (mapping two features to one) if they turn out to be important indicators of performance.

Finally, we added some features to represent other potentially important quantities. These features include the time of day (including features for morning and evening rush hours), the day of the year (mapping to the season), and the elapsed time since the device was plugged in (for 'warming up' effects).

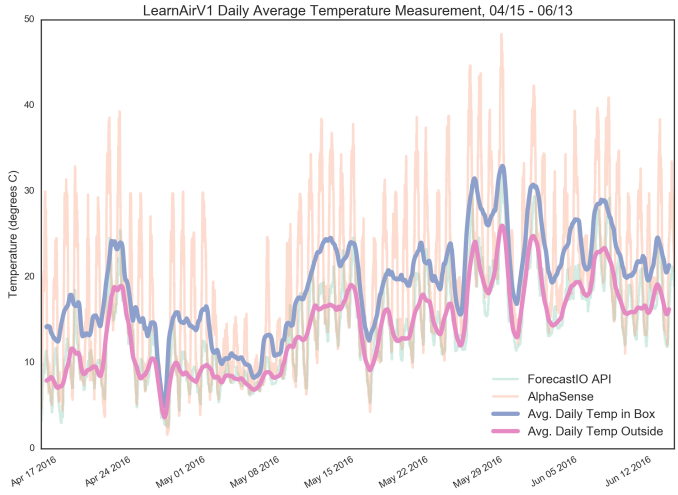


Figure 28: Temperature Inside vs. Outside the Device during Test Period

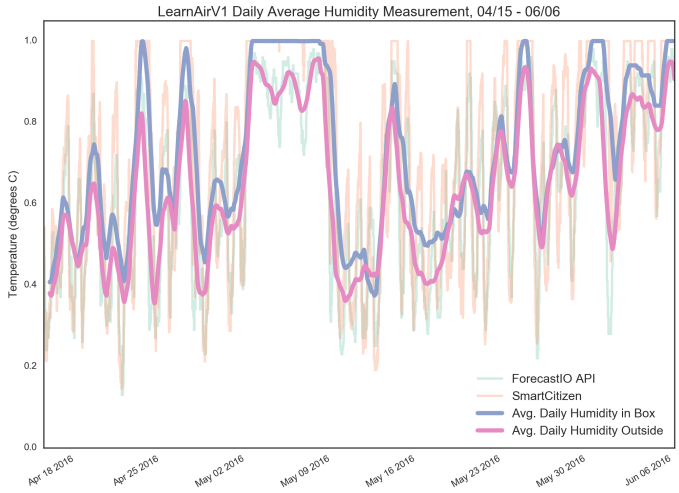


Figure 29: Humidity Inside vs. Outside the Device during Test Period

In general, some small subset of features was removed for each training session. For instance, the higher quality Alphasense CO sensor was removed as a feature when training the less capable SmartCitizen CO sensor. (Training with this feature gives incredibly high accuracy at predicting failure, because it is effectively training itself with the answer.) By training with comparable or cheaper sensors, we can assess the likelihood of a cheap system working predictably.

In most cases, the EPA reference black carbon sensor data was left as a feature– while this is not a feasible measurement for a portable, cheap device, it is still useful to know if black carbon is a strong predictor of a sensor's failure. This knowledge allows us to infer something about how the sensor is failing, and what type of sensor we might want to pair it with. With this technique, the machine learning process is not evaluative of a current system's success–

instead, it works as an exploratory tool that might inform a potential system design.

The next page includes a list of the main features. Processed, calibrated, and other derived features extend this list to the complete set. For histogram plots of many of the main features (made using Weka), see Appendix D.

Table 1: Machine Learning Features used to Predict Sensor Accuracy

Feature	Type	Description
AlphaSense #1 Work Electrode	Continuous	raw signal from alphasense sensor
AlphaSense #1 Aux Electrode	Continuous	raw signal from alphasense sensor
AlphaSense #2 Work Electrode	Continuous	raw signal from alphasense sensor
AlphaSense #2 Aux Electrode	Continuous	raw signal from alphasense sensor
AlphaSense #3 Work Electrode	Continuous	raw signal from alphasense sensor
AlphaSense #3 Aux Electrode	Continuous	raw signal from alphasense sensor
AlphaSense O_3	Continuous	calibrated signal from raw electrode readings
AlphaSense NO_2	Continuous	calibrated signal from raw electrode readings
AlphaSense CO	Continuous	calibrated signal from raw electrode readings
AlphaSense H_2S	Continuous	calibrated signal from raw electrode readings
AlphaSense Temperature	Continuous	raw signal from alphasense temp sensor
Wind Pressure Reading	Continuous	raw signal from pressure sensor
Corrected Wind Reading	Continuous	conditioned pressure signal to relate to wind
Sharp Dust Reading	Continuous	raw signal from optical particulate sensor
SmartCitizen CO Voltage	Continuous	raw signal from smartCitizen CO sensor
SmartCitizen NO_2 Voltage	Continuous	raw signal from smartCitizen NO_2 sensor
SmartCitizen Noise Voltage	Continuous	raw signal from smartCitizen Piezo mic
SmartCitizen Humidity Voltage	Continuous	raw signal from smartCitizen SHT15
SmartCitizen Temperature Voltage	Continuous	raw signal from smartCitizen SHT15
SmartCitizen Humidity	Continuous	conditioned smartCitizen Humidity Measurement
SmartCitizen Temperature	Continuous	conditioned smartCitizen Temperature Measurement
SmartCitizen Light Reading	Continuous	raw signal from smartCitizen light sensor
SmartCitizen Solar Panel Charge	Continuous	raw signal from smartCitizen sensor
EPA Sensor Wind Direction	Continuous	calibrated, accurate EPA reference measurement
EPA Sensor Wind Speed	Continuous	calibrated, accurate EPA reference measurement
EPA Sensor Black Carbon	Continuous	calibrated, accurate EPA reference measurement
ForecastIO Apparent Temperature	Continuous	calibrated api call measurement
ForecastIO Cloud Cover	Continuous	calibrated api call measurement
ForecastIO Dew Point	Continuous	calibrated api call measurement
ForecastIO Humidity	Continuous	calibrated api call measurement
ForecastIO Precipitation Intensity	Continuous	calibrated api call measurement
ForecastIO Precipitation Probability	Continuous	calibrated api call measurement
ForecastIO Pressure	Continuous	calibrated api call measurement
ForecastIO Temperature	Continuous	calibrated api call measurement
ForecastIO Visibility	Continuous	calibrated api call measurement
ForecastIO Wind Bearing	Continuous	calibrated api call measurement
ForecastIO Wind Speed	Continuous	calibrated api call measurement
ForecastIO Clear Night	Boolean	calibrated api call measurement
ForecastIO Clear Day	Boolean	calibrated api call measurement
ForecastIO Partly Cloudy Day	Boolean	calibrated api call measurement
ForecastIO Partly Cloudy Night	Boolean	calibrated api call measurement
ForecastIO Cloudy	Boolean	calibrated api call measurement
ForecastIO Rainy	Boolean	calibrated api call measurement
ForecastIO Foggy	Boolean	calibrated api call measurement
ForecastIO Windy	Boolean	calibrated api call measurement
Morning Hours	Boolean	created field to correspond to the morning
Afternoon Hours	Boolean	created field to correspond to the afternoon
Evening Hours	Boolean	created field to correspond to the evening
Morning Rush Hours	Boolean	created field to correspond to the morning rush
Lunch Hours	Boolean	created field to correspond to the lunch
Evening Rush Hours	Boolean	created field to correspond to the evening rush
Day Hours	Boolean	created field to correspond to the day
Night Hours	Boolean	created field to correspond to the night
Outside and Inside Box Temp Differential	Continuous	Difference between temp out/in box
Minutes Since Plugged In	Continuous	to help quantify initial terrible readings as sensor settles
Day of Year	Continuous	day resolution proxy for seasonality

General Trends in the Data

Before diving into each measurement in detail, we'll look at the overall trends in the data collected by the MassDEP reference sensors relative to our learnAir sensors. As we can see in Figure **??**, there are four reference measurements taken, each with slower moving trends as well as clear transient spikes.

We expect transient events in CO, NO_2, and black carbon, the direct by-products of combustion. As noted in the background section, these pollutants frequently vary with rapid time-scales, and their transient phenomena have been well characterized. O_3 is also known to have transient events, despite the more complicated reactions that drive its creation. This behavior has been modeled, and studies of other cities have shown similar magnitude/duration transient behavior in measured O_3 levels. It is suspected that meteorological conditions and high NOx are to blame in rapid ozone formation, as well as large concentrations of other highly reactive VOCs. [81, 82, 83]

Zooming in on these transient events reveals their duration– NO_2 transients fall in the minute-duration range, while O_3 and CO tend to sustain for about half an hour at a time. Black carbon is integrated hourly, so true transient behavior cannot be observed.

The variability of pollutant concentration as a function of temperature, wind speed, emission flow rate, emission concentration level, and flue height (which is modeled itself on a variety of parameters) greatly affects dispersion characteristics. The distance from roadside and height of these sensors makes it quite plausible that transient events will disperse before reaching a sensor and occur over longer time-scales. [23] The lack of clear relationships between time-of-day and these transients– as well as the relative independence of each sensor– is not surprising. It does appear a subset of NO_2, CO, and O_3 transient events occur close in time, and likely share a causal link (as would be expected, since they are associated with the same sources– likely heavy diesel traffic for this installation).

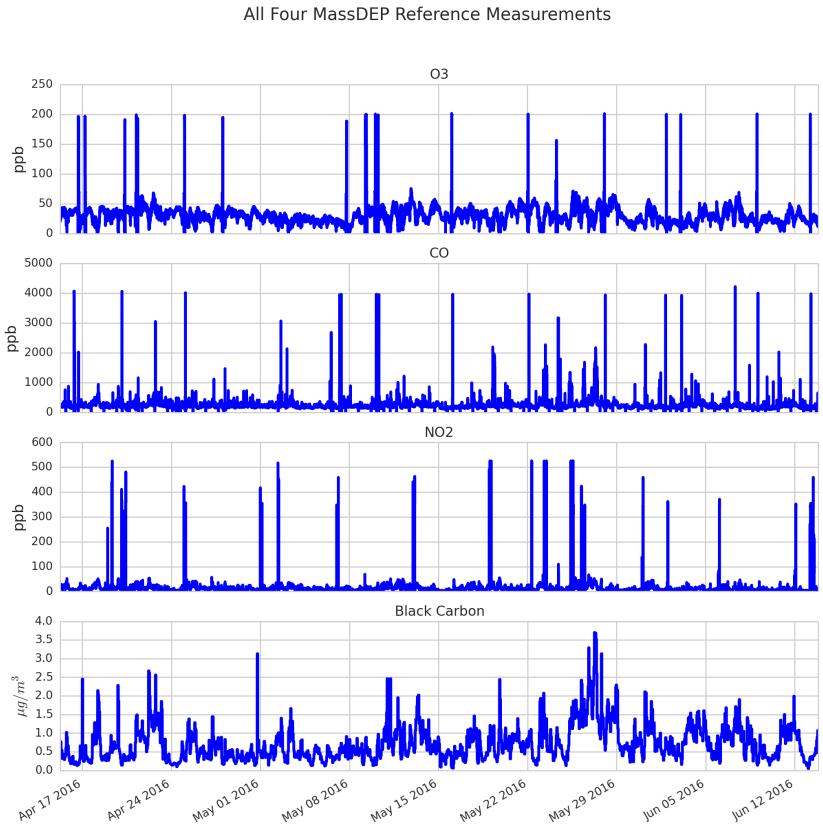


Figure 30: Reference Sensors Measurements During Test Period

Generally, we see an inability to capture transient phenomena with the sensors we tested. The cheaper NO2 and CO sensors on the SmartCitizen Kit showed no transient behavior. The Alphasense NO2 also struggled to capture any transient events. The AlphaSense CO sensors reported a handful of small transients (0.5-1.5 ppm) that generally appear to match up with real transient events– however, these transients were generally much smaller in magnitude than the actual, and most transients were still uncaptured. The AlphaSense O3 sensor similarly captured a handful of transient events, with a closer match to the appropriate magnitude. Unfortunately, it also missed about half of the transient events.

These differences are interesting, and can potentially be explained in a few ways. The SmartCitizen sensors (as expected) simply aren't of good enough quality to be sensitive and reactive enough to measure these differences. They are speced for much higher concentrations at much lower resolutions. The Alphasense sensors seem to have

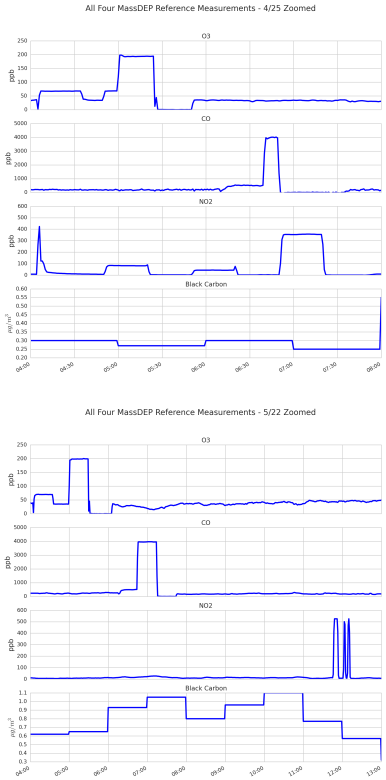


Figure 31: Two Example Transient Events Measured by Reference Sensors, 4/25 and 5/22

the ability to respond in the CO and O_3 case. These transients are actually quite long events, so there is no time-constant issue with the sensor physics. Missed peaks and/or mis-measurments are likely due to airflow effects– the reference sensor is actively pumping air through it, while the learnAir sensor is passively receiving airflow with some bias in its directionality. The successful capture of a few transients suggest that the sensor physics are fundamentally up to the task.

NO_2 transient events were much shorter duration events, and occur in the 0-400 ppb range. The response time for a few of the narrower events could come into play, as the peaks clearly bump up against our 1-minute resolution (15 second rise time for 500 ppb); however, this is unlikely to be a primary factor, especially for some of the several minute transient events. At the end of the day, these are slight changes in concentration (a few hundred ppb) occurring at much faster timescales than CO and O_3, and it seems that the combination of these effects in conjunction with airflow made it impossible to detect. It is worth further investigation to understand whether or not the sensor itself is capable of detecting these rapid events.

SmartCitizen CO

One SmartCitizen CO sensor was tested against the EPA reference. It was 1 month old at the time of installation, and ran for 52 days (from 4/15 - 6/6 2016) with two 40 minute service interruptions. This test gave 74,961 samples of minute resolution data for this machine learning task.

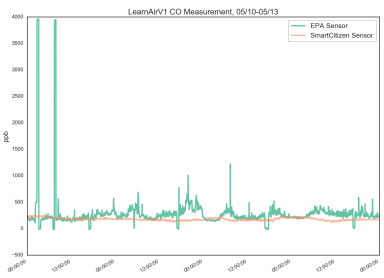


Figure 32: SmartCitizen CO Raw Data (orange) vs. EPA reference (green)

Pre-processing

The SmartCitizen CO sensor data comes uncalibrated, as a mV value that should correlate to CO concentration. The first step was to run a bounded LMSE minimization on the data in order to scale and offset it appropriately to match the real data. You can see the final result of such a scaling in Figure 33. You'll notice that the LMSE minimization basically scaled the sensor values down to a minimal amount of variation. This suggests that the sensor data itself is relatively useless in this context, which is relatively unsurprising given its working range and the near constant <1 ppm exposure.

Interestingly, the SCAQMD (South Coast Air Quality Management District) showed good correlation for 5-minute average, 500-1000 ppb range over their ten day co-location study. Our measurements were a few hundred ppb lower on average. It's also unclear which version of the Smart Citizen sensor SCAQMD tested, as Smart Citizen released an updated version of their board with completely different CO and NO2 gas sensors. It appears likely, given the test date, that they were using a different (though similarly priced) sensor module entirely.

Machine Learning

Machine learning on this data will tell us effectively nothing about the sensor's accuracy, since the sensor's correlation to the correct values is so poor. We don't need machine learning to see that the sensor has failed to predict meaningful values. Instead, applying machine learning here reduces to a comparison of the real CO concentration to a (more or less) constant baseline– this means our machine learning techniques simply attempt to predict when transients and elevated levels of CO occur at this location using metrological and other sensor data.

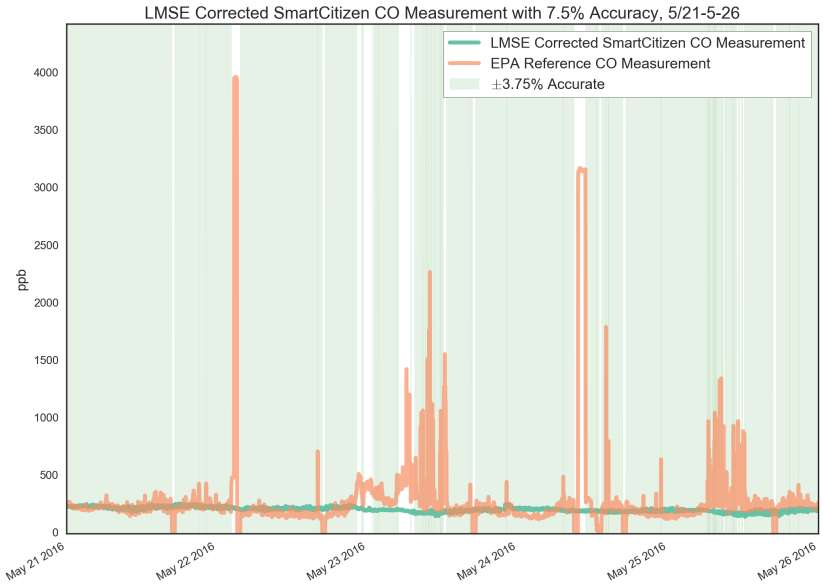


Figure 33: SmartCitizen CO with 7.5% Accuracy Threshold

In this case, the threshold for an 'accurate' reading made by the SmartCitizen CO sensor was set at 7.5% (or ±3.75%) of the full range of values detected for the actual CO levels, 315 ppb (±157.5 ppb). Figure 33 shows the Smart Citizen values against the reference, with

Error Rates for SmartCitizen CO with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.09	0.09	0.09	0.09
min	0.08	0.03	0.09	0.08
max	0.09	0.12	0.09	0.11

Table 2: Error Rates for Predicting SmartCitizen CO Accuracy with Logistic Regression

'accurate' measurements highlighted with a green background. The 'inaccurate' readings are the peaks and large diversions from the baseline.

As with all of the machine learning models, scikit learn was used to conduct a parameterized search for the optimal logistic regression was conducted using parameters C = [0.001, 0.1, 10, 1000] and penalty-type=['L1', 'L2'] with a 2-fold cross-validation. This covers a reasonable amount of parameter space for the minimal 16 rounds of training and validation (1-2 hours on my i5 laptop). For this test, an L1 penalty and C=10 regularization term gave the best ROC_AUC score of 0.82.

We see similar error rates of about 9% error in our predictions regardless of whether we chunk or shuffle the data, or whether we use the full set of 150 features or just a subset of the top fifteen. The average confusion matrix (five shuffled trained sets validated on a new 1/5 of the full data-set each time) is shown in Table . The confusion matrix for the chunked set has similar values.

		Predicted Values	
		0	**1**
Actual Values	**0**	197.0	1187.4
	1	108.6	13499.2

Table 3: Average SmartCitizen CO Confusion Matrix w/Shuffled K-Fold

Though these conditions look the same based on average error rate and confusion matrix, when we examine the confidence of our predictions we see some dramatic differences in the shuffled vs. chunked cases. From Figure 34 we can see the shuffled tests resulted in strong, consistent AUC-ROC scores between 0.81 and 0.83, while the chunked version resulted in unconvincing, highly variable results from 0.43 (worse than uneducated guessing) to 0.76. When using only the top 15 features from a random tree algorithm, we see similar trends, however our accuracy drops from slightly above 0.8 to slightly about 0.7 for the shuffled case (see Appendix D for a list of the top features selected in this way, and a graph of ROC results for the same algorithm using just those features).

Table 4 shows the top features for predicting CO variations away from the baseline, as determined by seven different feature reduction techniques.

Table 4: Top Features for Predicting SmartCitizen CO

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
bkcarbon	1	0	0	1	0.58	0.28	0.93	0.54
avg_60_bkcarbon	0.98	0	0	0.25	0.53	0.15	0.83	0.39
evening	0.17	0	0.07	0.19	0.82	0.18	1	0.35
avg_1440_bkcarbon	0.57	0	0	0.37	0.53	0.44	0.54	0.35
humidity_box_differential	0.1	0	0.01	0.22	1	1	0.02	0.34
afternoon	0.17	0	0.07	0	0.84	0.19	1	0.32
avg_60_forecastio_humidity	0.01	0	0.01	0.12	1	1	0	0.31
temp_sck_box_differential	0.09	0	0	0.45	0.84	0	0.73	0.3
Solar Panel (V)	0.05	0	1	0	0.77	0	0	0.26
avg_720_bkcarbon	0.65	0	0	0.26	0.27	0.14	0.43	0.25
forecastio_apparentTemperature	0.04	1	0	0.03	0.16	0	0.43	0.24
lmse_avg_30_scaled_arduino_ws	0	0	0	0.05	0.8	0.03	0.79	0.24
forecastio_clear-night	0	0	0.08	0.04	0.91	0.06	0.51	0.23
forecastio_partly-cloudy-day	0.06	0	0.08	0	0.91	0	0.55	0.23
forecastio_partly-cloudy-night	0.01	0	0.08	0	0.89	0.06	0.54	0.23
avg_30_scaled_arduino_ws	0	0	0.02	0.07	0.81	0	0.74	0.23
Noise (mV)	0.02	0	0	0.11	0.39	0.02	0.92	0.21
avg_720_lmse_scaled_sharpDust	0.03	0	0	0.15	0.55	0.22	0.52	0.21
derivative_avg_720_bkcarbon	0	0	0	0.12	0.63	0.25	0.45	0.21
daily_avg_sck_humidity	0.07	0	0	0.18	0.59	0.17	0.4	0.2
derivative_avg_360_lmse_as_no2	0	0	0	0.11	0.55	0.73	0	0.2
derivative_avg_1440_bkcarbon	0.02	0	0	0.14	0.64	0.02	0.58	0.2
evening_rush	0.13	0	0	0.04	0.49	0.1	0.54	0.19
avg_60_forecastio_pressure	0.07	0	0	0.26	0.41	0.01	0.6	0.19

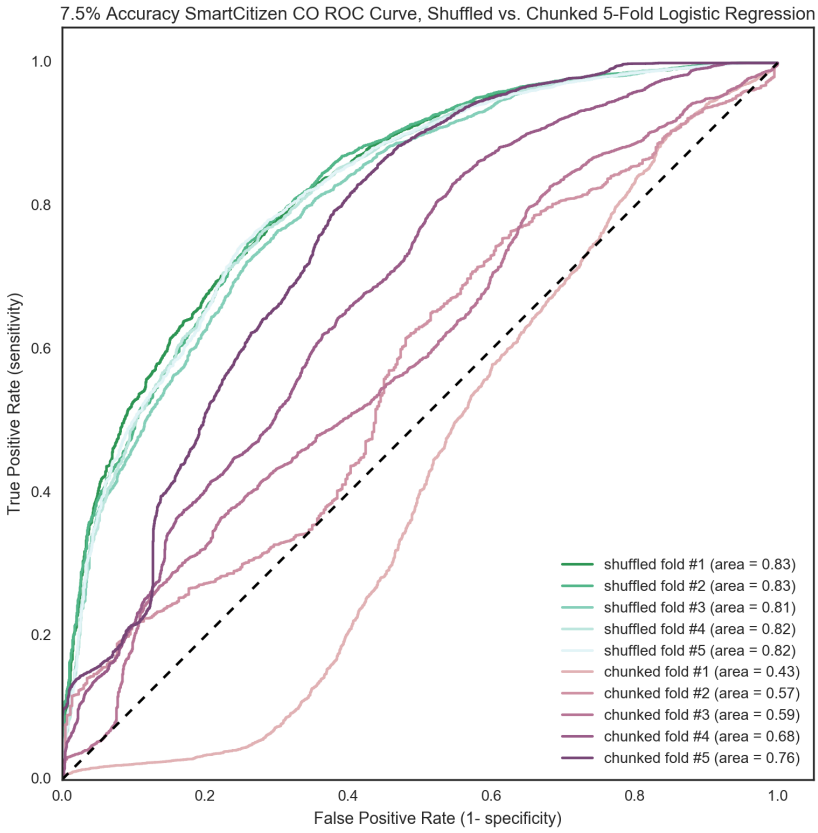


Figure 34: SmartCitizen CO ROC Curve

The top features– combined with our ROC results– give us insight into the usefulness of this algorithm for predicting transient fluctuations in CO. The large divergence between shuffled and chunked data suggests we have not collected enough data yet to make seasonally-agnostic predictions in CO. However, the promising results in our shuffled data lead us to believe it is possible to make fairly robust predictions (within or accounting for the season) with enough data collection. The relationships we find in our shuffled results are consistent and reliable across test sets.

The top features include black carbon readings, time of day designations (like evening or night), windspeed measured at the sensor, and temperature. These are in line with expectation– as a similar combustion byproduct, we would expect black carbon and CO to be closely related. Heavy traffic (the main source of both black carbon and CO) has predictable time-of-day patterns. Temperature and windspeed have been directly correlated to changes in CO concentration, given a constant, predictable source. Furthermore, we'd expect our prediction to be based on a complex relationship of many underlying features.

The difference in results from the full feature set to the reduce set corroborates this assumption.

In our test, the SmartCitizen CO sensor provided no meaningful data when compared and fit against the FRM reference. This changed our machine learning task from a 'predict sensor accuracy' problem to a 'predict CO transient' one. Our results indicate a strong likelihood of making good predictions of transient events for CO with a high quality black carbon sensor as part of the device. However, these predictions are complex, and require a lot of diverse sensor data to tease out a strong prediction. Furthermore, this relationship appears to be seasonally dependent, and thus an extensive training set– longer than the two month season change we captured here– is necessary to predict CO transient behavior with good accuracy.

See Appendix D for more plots outlining the LMSE pre-processing steps, other accuracy thresholds we attempted, a plot of the SmartCitizen data with a correct/incorrect prediction overlay, and the Random Tree reduced-feature selection table and corresponding ROC curves.

SmartCitizen NO2

One SmartCitizen NO2 sensor was tested against the EPA reference. Like the SmartCitizen CO test, it was 1 month old at the time of installation, and ran for 52 days (from 4/15 - 6/6 2016) with two 40 minute service interruptions. It gave 74,961 samples of minute resolution data.

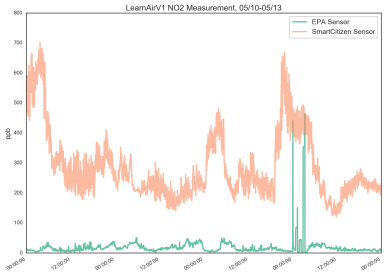


Figure 35: SmartCitizen NO2 Raw Data

Pre-processing

Like the SmartCitizen CO data, the NO2 sensor comes uncalibrated, as a mV value. The first step was to run a bounded LMSE minimization on the data in order to scale and offset it appropriately to match the real data. You can see the final result of such a scaling in Figure 36. Once again, the LMSE minimization scaled the sensor values down to a minimal amount of variation. This suggests that the sensor data itself is relatively useless in this context, which is relatively unsurprising given its working range and the near constant <1 ppm exposure. This agrees with spurious test findings for three SmartCitizen NO2 sensors tested by SCAQMD.

This case is similar to the SmartCitizen CO sensor, as machine learning will tell us nothing of the sensor's accuracy. It reduces to a different problem, instead– predicting NO_2 transients and elevated levels based on metrological and other sensor data.

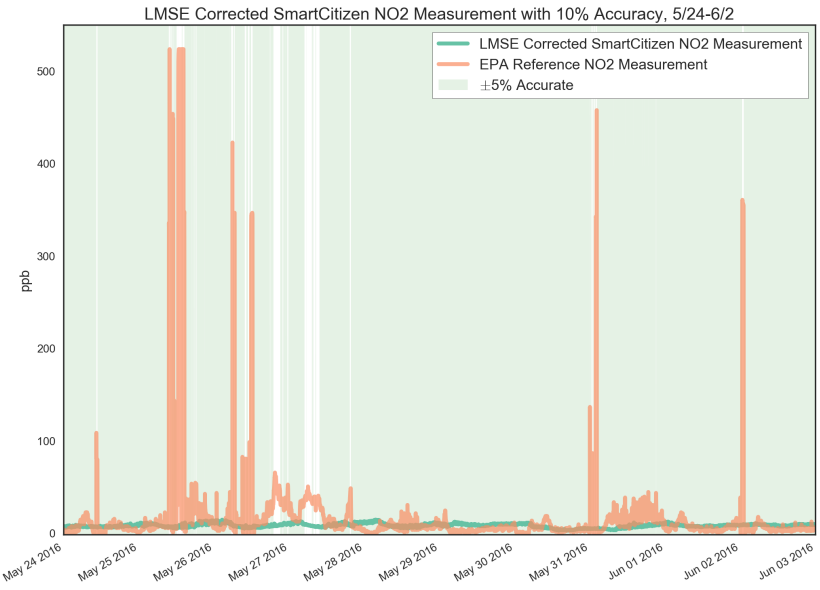


Figure 36: SmartCitizen NO_2 with 10% Accuracy Threshold

The threshold for accuracy was set at 10% (or ±5%) of the full range of values detected for the actual NO_2 levels, 50 ppb (±25 ppb). Figure 36 shows the Smart Citizen values against the reference, with 'accurate' measurements highlighted with a green background. The 'inaccurate' readings are the peaks and large diversions from the baseline.

As with all of the machine learning models, scikit learn was used to conduct a parameterized search for the optimal logistic regression was conducted using parameters C = [0.001, 0.1, 10, 1000] and penalty-type=['L1', 'L2'] with a 2-fold cross-validation. An L1 penalty and C=1000 regularization term gave the best ROC_AUC score of 0.91.

We see very low error rates of 3-4% on average with our predictions, regardless of the size of our feature set and whether the validation was randomized or chunked. This is corroborated by very strong performance in the average confusion matrix (Table). Our ROC curves show tight agreement for the shuffled test sets of 0.90-0.92

Error Rates for SmartCitizen NO2 with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.03	0.05	0.03	0.04
min	0.03	0.02	0.03	0.02
max	0.03	0.08	0.04	0.06

Table 5: Error Rates for Predicting SmartCitizen NO2 Accuracy with Logistic Regression

AUC-ROC, while the chunked version lags behind slightly with scores from 0.69-0.83. These results show evidence of extremely strong predictive power, though they hint that we haven't collected quite enough data to completely account for seasonal variation (even though we're doing well enough with the chunked test sets to predict with good confidence).

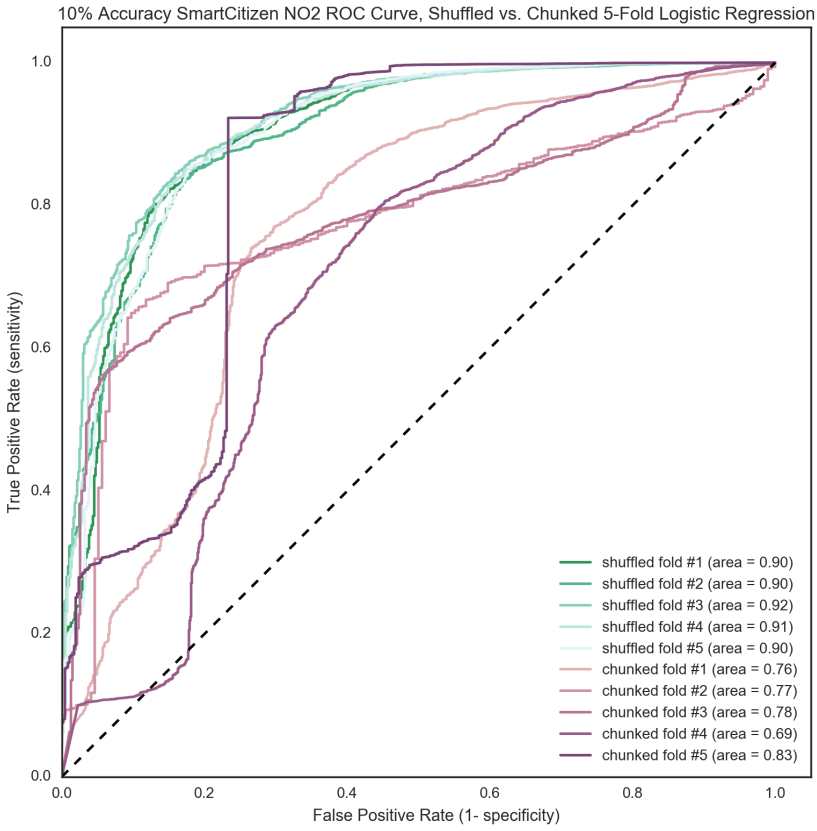


Figure 37: SmartCitizen NO2 ROC Curve

We see similar error rates with all features as for a subset of the top fifteen. Examining the AUC-ROC shows that when training only with the top features, the score drops from 0.9 to 0.8 – a substantial drop, but proof that the top fifteen account for a very strong prediction on their own.

The top features are similar to those that predicted the CO transients, but with a stronger relationship– black carbon is a leading indicator, as well as humidity, the hour of the day, and CO level. These all make sense as correlates– CO and black carbon are produced by traffic, which has a predictable pattern with the time of day. Table 6) shows the top features given seven feature selection algorithms.

Table 6: Top Features for Predicting SmartCitizen NO2

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
bkcarbon	1	0	0	1	0.63	0.51	0.85	0.57
daily_avg_sck_humidity	0.11	0	0	0.32	0.67	1	0.76	0.41
hour_of_day	0.07	0.92	0	0.1	0.42	0.02	1	0.36
avg_60_bkcarbon	0.92	0	0	0.08	0.59	0.25	0.69	0.36
derivative_avg_1440_lmse_calib_as_co	0.06	0	0	0.05	0.62	0.38	1	0.3
Solar Panel (V)	0.03	0	1	0	0.88	0	0	0.27
lmse_sck_co	0.01	1	0	0.02	0.85	0	0	0.27
derivative_avg_1440_bkcarbon	0.05	0	0	0.07	0.71	0.1	0.99	0.27
humidity_box_differential	0.02	0	0.04	0.11	1	0.61	0	0.25
forecastio_partly-cloudy-night	0.03	0	0.16	0.02	0.95	0.11	0.42	0.24
avg_60_forecastio_humidity	0	0	0.04	0.06	1	0.61	0	0.24
day	0.07	0	0	0	0.97	0.05	0.51	0.23
night	0.07	0	0.01	0	0.98	0.05	0.43	0.22
avg_60_forecastio_precipProbability	0.04	0	0	0	0.6	0.49	0.44	0.22
daily_avg_forecastio_humidity	0.03	0	0	0.05	0.66	0.48	0.24	0.21
forecastio_rain	0.03	0	0.16	0	0.9	0.25	0.04	0.2
forecastio_humidity	0	0	0	0	0.64	0.72	0	0.19
forecastio_clear-night	0.02	0	0.16	0	0.93	0.03	0.1	0.18
forecastio_fog	0	0	0.16	0	0.9	0.21	0	0.18
forecastio_wind	0.01	0	0.16	0	0.92	0.18	0	0.18
avg_720_bkcarbon	0.47	0	0	0.05	0.54	0.07	0.12	0.18
avg_30_ws	0.13	0	0	0.08	0.45	0.02	0.48	0.17
avg_15_derivative_sck_temperature	0	0	0	0.04	0.63	0.5	0.01	0.17
avg_720_lmse_scaled_sharpDust	0.01	0	0	0.16	0.58	0.41	0	0.17

Once again, there seems to be strong evidence that NO2 transients are highly predictable if there are other, good quality sensors that measure related phenomena (like black carbon and CO) to rely on. The relationship is a complex one that requires a larger feature set and a lot of training data for optimal performance, but it appears that even in the short two month period we were able to train a model that would work well under realistic conditions.

		Predicted Values	
		0	**1**
Actual Values	**0**	142.0	427.0
	1	52.6	14370.2

Table 7: Average SmartCitizen NO2 Confusion Matrix w/Shuffled K-Fold

Sharp Dust Sensor

One Sharp optical particulate sensor was tested against the EPA black carbon reference. It was 1 month old at the time of installation, and ran for 59 days (from 4/15 - 6/13 2016) with two 40 minute service interruptions. This test gave 1,431 samples of hour resolution data.

Pre-processing

The Sharp sensor is a common choice for cheap air quality sensing projects, but is generally known as a simple device that can struggle in outdoor conditions. The signal from the sensor is a raw light magnitude, so to convert it to $\mu g/m^3$ requires an inversion and basic scaling. After inversion, a LMSE optimization was applied against the reference sensor. Instead of a straight LMSE algorithm, though, the cost function was modified to include only the values that were within 5% of the reference signal. This eliminates minimization errors due to regions where the sensors are mis-reading and mismatched. This technique clearly out-performed the standard LMSE techniques tried (overall LMSE, LMSE without 'outliers' more than one standard deviation away, etc). This method worked well, and the sensor clearly followed the general trend of the EPA reference in many cases. See Figure 38 for an example.

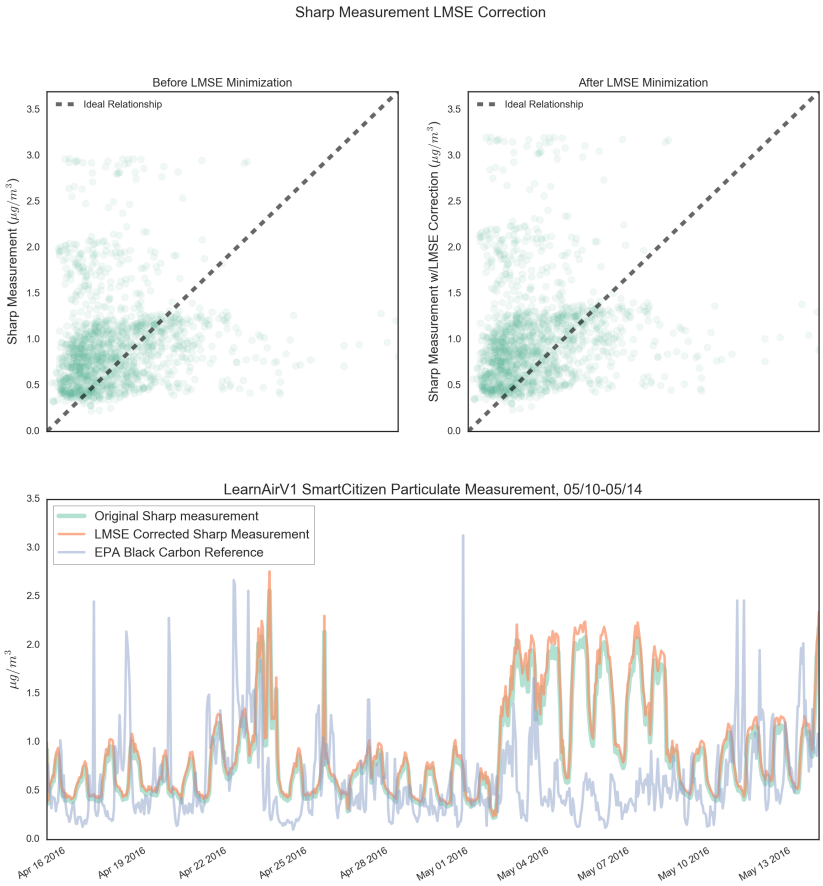


Figure 38: Sharp Particulate LMSE Calibration

The reference sensor, for its part, labels readings in the following

way: a '10am reading' of EPA sensor is recorded on filter paper from 10a-10:50a, then measured with an optical attenuation technique from 10:50-11a. Thus, we averaged the 50 minute readings starting on the hour and threw away the last ten minutes. After comparison, we added features with 6-, 12-, and 48- hour averaging. The 48 hour measurement was also run through a LMSE process and used for machine learning (see Figure 39).

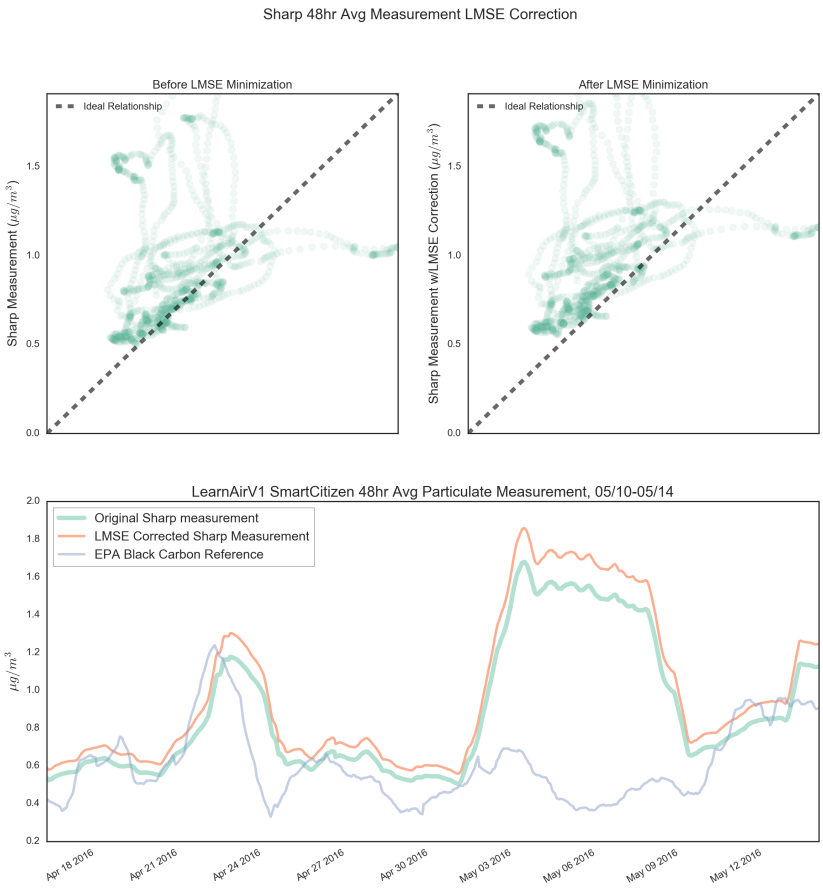


Figure 39: 2 day Average Sharp Particulate LMSE Calibration

Machine Learning

Machine learning on this data can provide us with useful insights about our ability to predict when the sensor works and when it does not. The main test was run on errors using a 30% tolerance– 0.56 $\mu g/m^3$. Besides running our algorithm on the standard LMSE minimized signal, we also ran it using tried it using a 15% tolerance ($0.23\mu g/m^3$), and 48-hr average with a 30% tolerance based on its max value ($0.29\mu g/m^3$). Figure 40 shows the main comparison with a

Error Rates for Sharp Sensor with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.17	0.22	0.22	0.25
min	0.14	0.18	0.20	0.14
max	0.20	0.29	0.24	0.36

Table 8: Error Rates for Predicting Sharp Accuracy with Logistic Regression

30% accuracy overlay.

Since this test uses hourly data, it is much faster to train and test a model. It also means we have much less data to draw strong conclusions with. In this case, we used a complete five fold validation to optimize our logistic regression parameter search of C = [0.001, 0.1, 10, 1000] and penalty-type=['L1', 'L2']. We found the best parameters were C=1 with an L1 penalty. For the tighter tolerance, an L2 penalty was more successful, and for the average values, C=10 worked best.

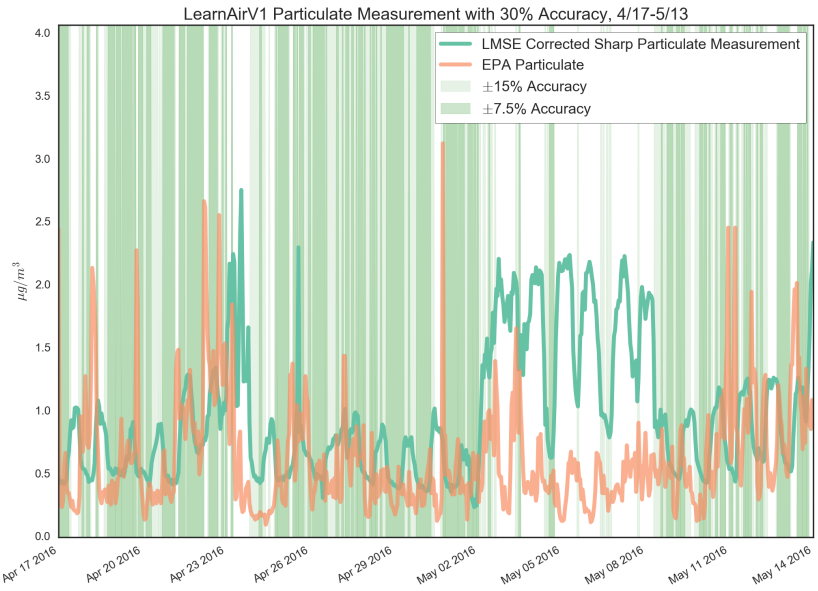


Figure 40: Sharp Particulate with 30% Accuracy Threshold

Error rates in Table 8 show that we have a large difference between shuffled and chunked techniques, indicating we have no captured enough data to make fully seasonally robust predictions. The error rates are also relatively high.

Despite the high error rates, our AUC-ROC curve shows strong results (0.86 to 0.88 shuffled, 0.71 to 0.95 chunked)– this suggests we're good at reporting the strength of our prediction (i.e., when we're wrong we reliably report the likelihood of a correct prediction close

		Predicted Values	
		0	**1**
Actual Values	**0**	58.0	35.2
	1	14.0	179.0

Table 9: Average Sharp Particulate Confusion Matrix w/Shuffled K-Fold

to 50%, whereas when we're right we reliably report a likelihood closer to 100%).

For 48-hour average data, we half the error rate in the shuffled case (down to 0.08), though the chunked case remains nearly the same. (As a reminder, this is with a tighter absolute error tolerance as well, of $\pm 0.15\ \mu g/m^3$ instead of ± 0.28). We see dramatic results in the ROC curve for the shuffled case– an average of 0.98 AUC-ROC. These are very promising indicators that we can be extremely reliable in our prediction of certainty for each reading, especially with the averaged data.

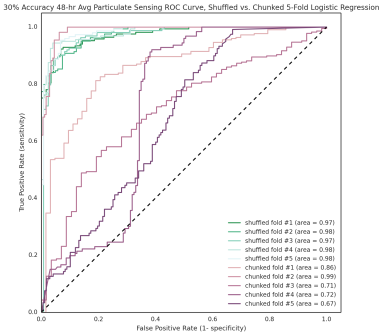
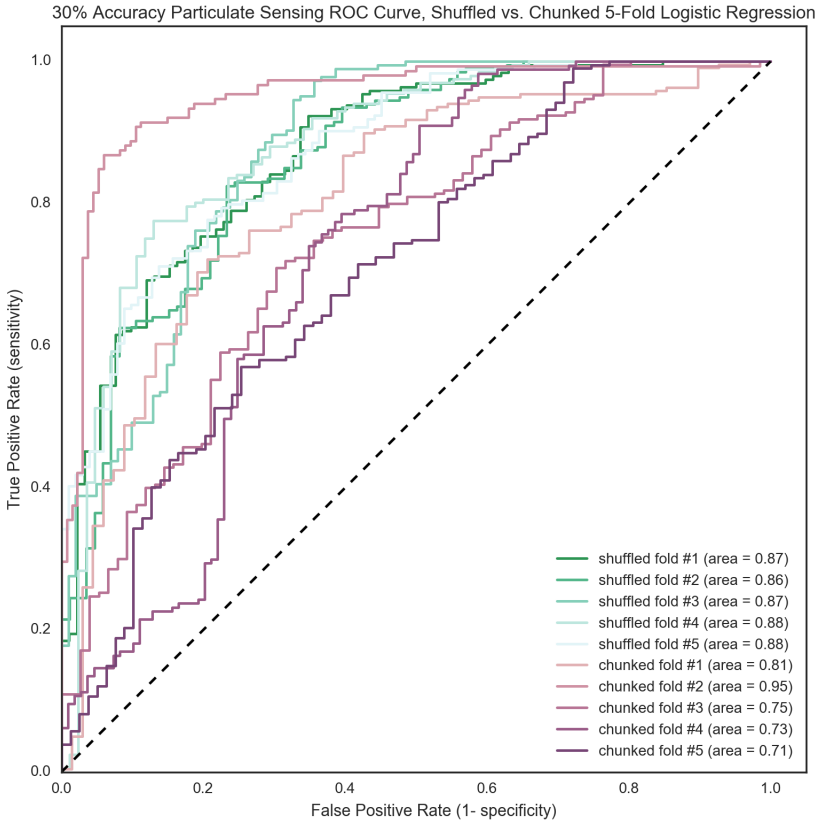


Figure 41: 48-hour Average Sharp Particulate ROC

Figure 42: Sharp Particulate ROC Curve



Our top 15 features do a relatively strong job on their own with these predictions, giving AUC-ROC values of 0.80-0.83 and 0.68-0.94 for the shuffled and chunked cases. The top features for predicting particulate are derived from the Sharp sensor itself– certain ranges of the sensor are more prone to error than others (high readings are less reliable). The other top features (Table 10) make sense given the physics of the device– changes in NO_2 seems to be a good indicator of sensor performance, as well as humidity level. (We would expect

Table 10: Top Features for Predicting Sharp Particulate

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
sharpDust	1	0.65	0	1	0.81	0.03	1	0.64
lmse_scaled_sharpDust	1	0	0.03	0.65	0.83	0	0.98	0.5
scaled_sharpDust	1	0	0.02	0.65	0.83	0	0.91	0.49
avg_12_scaled_sharpDust	0.88	0	0	0.06	0.49	0.77	0.55	0.39
derivative_lmse_avg_15_as_no2	0	0	0.99	0	0.97	0.15	0.02	0.3
derivative_avg_15_lmse_as_no2	0	0	1	0	0.97	0.15	0.01	0.3
derivative_avg_48_scaled_sharpDust	0.01	0	0.99	0.01	1	0.01	0	0.29
derivative_lmse_avg_48_scaled_sharpDust	0.01	0	0.89	0.03	1	0.01	0	0.28
temp_as_box_differential	0.01	1	0	0.22	0.47	0.13	0.05	0.27
sck_humidity_saturated	0.11	0	0	0	0.61	1	0.01	0.25
Nitrogen Dioxide (kOhm)	0.18	0.06	0	0.02	0.72	0	0.67	0.24
daily_avg_sck_humidity	0.24	0	0	0.05	0.6	0.69	0	0.23
lmse_sck_no2	0.18	0	0	0.02	0.72	0	0.67	0.23
avg_48_scaled_sharpDust	0.45	0	0.02	0.01	0.88	0.18	0.07	0.23
lmse_avg_48_scaled_sharpDust	0.45	0	0.02	0.01	0.87	0.2	0.06	0.23
forecastio_wind	0	0	0	0	0.96	0.58	0	0.22
derivative_as_no2	0	0	0	0	0.99	0.57	0	0.22
03	0.1	0.25	0	0.06	0.14	0.01	0.85	0.2
derivative_Carbon Monxide (kOhm)	0	0	0.36	0.03	0.99	0.01	0	0.2
derivative_lmse_sck_no2	0	0	0.45	0.01	0.86	0	0	0.19
derivative_lmse_sck_co	0	0	0.31	0.02	0.98	0.01	0	0.19
daily_avg_forecastio_humidity	0.29	0	0	0.02	0.46	0.41	0.05	0.18
forecastio_rain	0.08	0	0	0	0.92	0.18	0	0.17
alphaTemp	0	0	0	0.01	0.75	0.39	0	0.16

higher humidity to result in fog and condensation around particles that disperse light and increase reading error).

These results are very promising. They indicate extremely strong prediction performance, especially when comparing long-term averages. They suggest we need to collect more data under more conditions before we are ready to predict across seasons. There is one caveat, however. In this test, the threshold for what is considered a 'correct' measurement is quite high, at 30% of the full range of observed values (0.6 $\mu g/m^3$ for the standard test). This is a useful resolution, however it is a constrained choice. When we cut this tolerance in half and retrain our models– see Appendix D for more details– the predictive quality starts to drop. The effect of this threshold on predictive reliability of our model can provide an indication of sensor precision. We can very accurately predict large errors with this sensor, but the inherent 'noise floor' of the device limits that predictive strength as we tighten our tolerance.

See Appendix D for more plots outlining the raw data, more information on the models with different accuracy thesholds and averaging, plots of the data with a correct/incorrect prediction overlay, and the Random Tree reduced-feature selection table and corresponding ROC curves.

AlphaSense CO

Two AlphaSense CO sensors were tested against the EPA reference. The first sensor was 2.5 years old at the time of installation, which ran for 38 days (from 4/15 to 5/23 2016 with one 40 minute service interruption). The second sensor was 2 months old at the time of installation, and ran for 21 days (from 5/23 - 6/13 2016). Our first test gave 55,589 minute-resolution samples to compare, our second test gave 30,150 samples.

These two tests give us a chance to compare manufacturer reliability and sensor aging effects. The two datasets can also be combined after calibration to draw overall conclusions about the AlphaSense model CO-A4 sensor.

Pre-processing

AlphaSense sensors give two main readings– from an auxilary and a working electrode. Additionally, they come with a calibration for sensitivity, as well as offset voltages for both electrodes. The calibration function combines all of these using a non-linear temperature-dependent term (which scales the working electrode term differently over a few temperature ranges, linearly interpolated).

While we used the provided calibration data to create one version of 'corrected data', we've found from talking with faculty in the Environmental Engineering group that it is not uncommon for these laboratory based calibrations to drift substantially. It is generally considered best practice to calibrate the sensors using co-location data. The datasheet calibration provides results as seen in Figure 43.

One of the first things to notice is what appears to be quantization errors in the data. This is not a result of the analog to digital conversion process on the Arduino– we have a 5V, 10-bit input, which gives

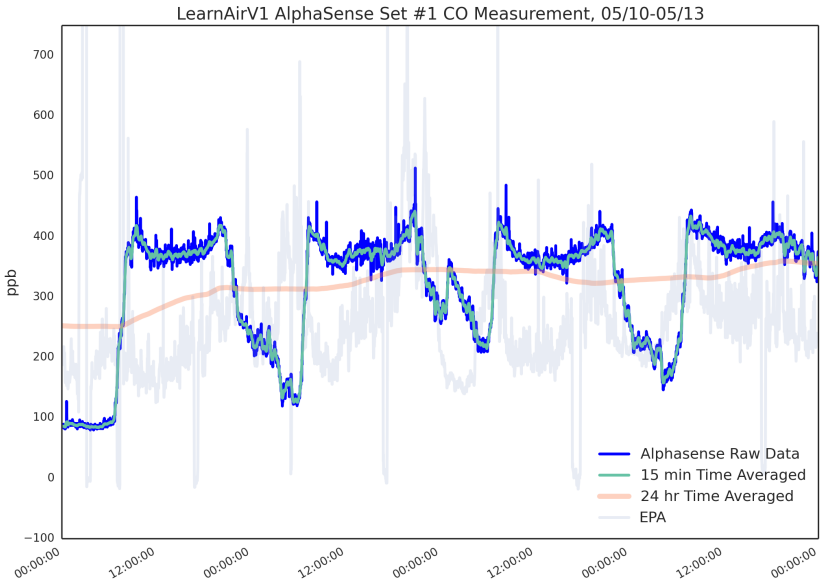


Figure 43: AlphaSense CO Sensor 1 Raw Data Zoomed

us 4.88 mV resolution. Our sensor has around 0.3 mV/ppb sensitivity, so we expect our ADC to quantize the signal in 16ppb steps. The AlphaSense analog frontend board for the CO sensor is only capable of a 20 ppb noise level, so the limiting factor is not our ADC. Examination of the raw Arduino data supports this assumption. This 'quantization' is actually the result of temperature dependent regimes in the calibration equation. At 10, 20, and 30 degrees Celcius we enter different linear temperature correction regimes– even though these are interpolated, the temperature dependence rapidly changes in a narrow span.

The proper way to 'correct' these calibrations is worthy of a paper by itself. Several techniques were attempted, mostly based on LMSE. We tried basic scaling/offset the datasheet calibrated data as well as scaling/offset the data considering only one standard deviation of the mean for each sensor (in order to remove outliers). We also attempted several more intuitive minimizations– for instance, solving for a new sensitivity and offset voltages (using the actual calibration equation) seeded with the datasheet values. Additionally, we tried minimizing by solving for new temperature dependence terms, in case the actual thresholds or scale factors were shifted slightly. Finally, we tried non-standard minimization strategies– setting thresholds for the values that are part of the cost minimization calculation, and using absolute error (since LMSE penalizes larger errors more, and we really want to ignore the larger errors without incentivizing a

cost function to create larger errors).

In general, the high frequency trends and sensitivity of the sensor are very in line with real pollutant concentration. The large, nearly step-wise variations due to temperature compensation seem to be the biggest issue with close tracking of the actual data. Most LMSE techniques aimed specifically at this temperature dependence completely eliminated the behavior– while this tracks the real signal better, it also gives a less dynamic signal with an unrealistic underlying assumption (that the sensor actually has *no* temperature dependence). With a more thorough understanding of how these sensors age and change, it may be easier to design an intuitive minimization for better calibration. For the purposes of this test (and the tests with other AlphaSense gas sensors), we minimized error by solving for new electrode offset voltages only. The results of this strategy for both the older and newer sensors can be seen in Figures 44 and 45.

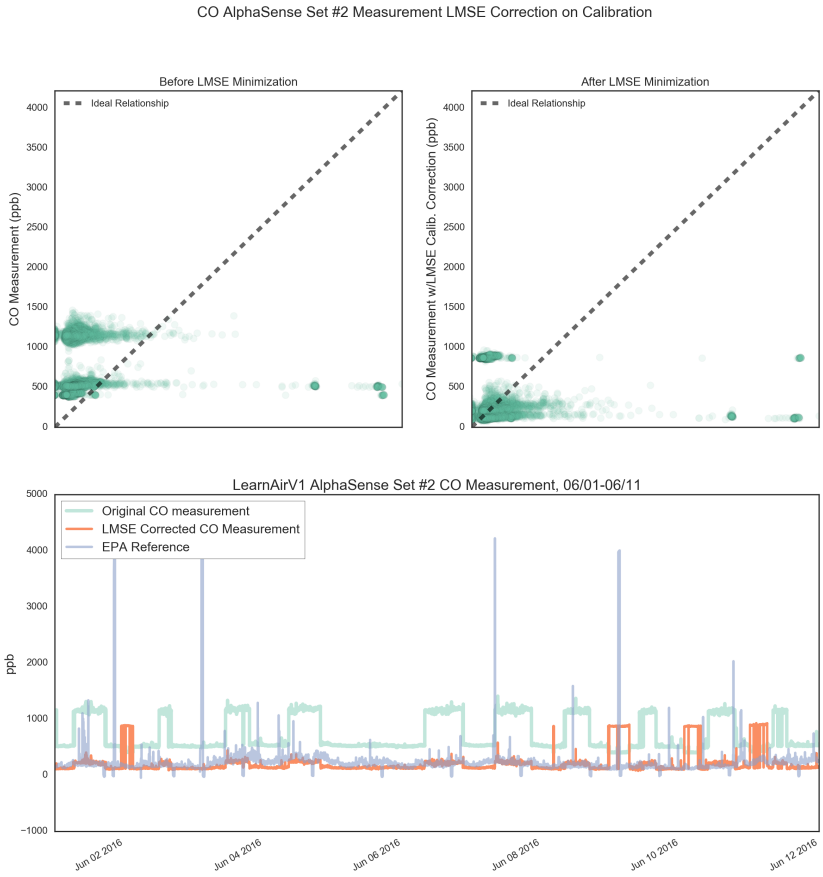


Figure 44: AlphaSense CO Sensor 2 after LMSE Calibration

Error Rates for CO Sensor 1 with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.18	0.21	0.20	0.20
min	0.17	0.17	0.19	0.12
max	0.18	0.28	0.20	0.28

Table 11: Error Rates for Predicting CO Sensor 1 Accuracy with Logistic Regression

Machine Learning

Both sensors were treated independently to start, and their output was compared. Agreement between the two can provide us with high confidence about underlying mechanisms. Both were optimized using a grid based parameter search of C = [0.001, 0.1, 10, 1000] and penalty-type=['L1', 'L2'] with a 2-fold cross-validation. Both gave the same optimum model using an L1 penalty and C=1000. The older sensor gave an AUC-ROC of 0.89, while the newer one gave 0.90. Figure 45 shows the comparison of the reference with our calibrated data, using a 5% tolerance (around ±100 ppb, a relatively ambitious tolerance).

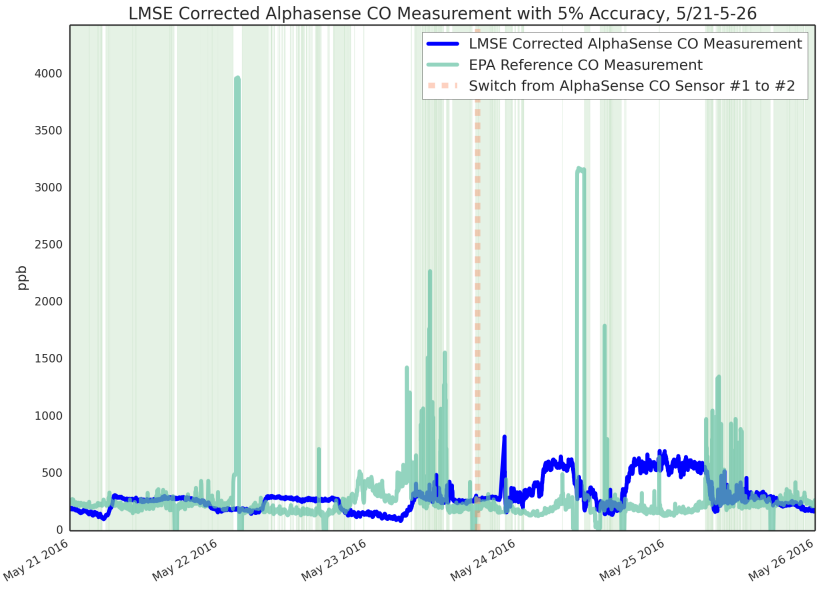


Figure 45: AlphaSense CO Sensor 1 and 2 with 5% Accuracy Threshold

The error rates of table 11 and 12 show relatively robust behavior from the entire feature set to just the top fifteen. The older sensor has slightly elevated error rates compared to the newer one, but it also saw more dynamic weather conditions. High variability in the chunked error rates suggests we haven't trained on enough data yet.

Error Rates for CO Sensor 2 with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.14	0.20	0.16	0.17
min	0.13	0.10	0.15	0.11
max	0.14	0.28	0.16	0.22

Table 12: Error Rates for Predicting CO Sensor 2 Accuracy with Logistic Regression

Despite the larger error rates, our AUC-ROC curves show reasonably good results– 0.89-0.90 for the shuffled, and 0.69-0.91 for the chunked. This is slightly better for the second, newer sensor (0.90-0.91 and 0.72-0.91). The results are thus quite promising in their predictive merit.

When we reduce the feature set to 15 features, we drop in accuracy to the mid- to high- 0.8s, but the agreement across sensors and across chunked and shuffled strategies is the highest we've seen so far. This stands out as a reasonably well executed model– it has verified predictive value, and it appears to agree with itself across season and across sensor. The top features– the sensor itself, NO_2, black carbon, humidity, temperature, and windspeed– are all common sense candidates for this type of sensor.

See Appendix D for more plots outlining the raw data, more information on the models with different accuracy thresholds and averaging, plots of the data with a correct/incorrect prediction overlay, and the Random Tree reduced-feature selection table and corresponding ROC curves.

		Predicted Values	
		0	**1**
Actual Values	**0**	4751.0	943.0
	1	1023.4	4400.4

Table 13: Average AlphaSense CO Sensor 1 Confusion Matrix w/Shuffled K-Fold

		Predicted Values	
		0	**1**
Actual Values	**0**	1326.2	623.0
	1	200.4	3880.4

Table 14: Average AlphaSense CO Sensor 2 Confusion Matrix w/Shuffled K-Fold

Table 15: Top Features for Predicting AlphaSense CO Sensor 1

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
as_co	0.95	0.29	0	1	0.64	0	0.51	0.48
lmse_calib_as_co	0.95	0	0	0.68	0.65	0	0.59	0.41
avg_60_forecastio_humidity	0.29	0	0.01	0.03	1	0.73	0.59	0.38
forecastio_wind	0	0	0.05	0	0.76	0.54	1	0.34
sck_temperature	0.92	0	0	0.02	0.98	0	0.44	0.34
alphaS2_work	0	1	0	0.02	0.25	0.01	1	0.33
alphaTemp	0.94	0	0	0	0.92	0	0.48	0.33
as_temperature	0.94	0	0	0	0.91	0	0.46	0.33
humidity_box_differential	0.14	0	0.01	0.04	1	0.73	0.37	0.33
avg_15_as_temperature	0.99	0	0	0.02	0.57	0.16	0.49	0.32
avg_720_lmse_scaled_sharpDust	0.02	0	0	0.03	0.54	1	0.68	0.32
Temperature (C RAW)	0.92	0.07	0	0.02	0.63	0	0.45	0.3
Solar Panel (V)	0.14	0	1	0	0.95	0	0	0.3
forecastio_temperature	0.68	0.09	0	0	0.87	0	0.37	0.29
avg_60_forecastio_temperature_c	0.7	0	0	0.03	0.97	0.09	0.25	0.29
evening	0.02	0	0.1	0.02	0.99	0.04	0.81	0.28
night	0.18	0	0.05	0	1	0.11	0.59	0.28
day	0.18	0	0.05	0	0.96	0.11	0.6	0.27
forecastio_temperature_c	0.68	0	0	0	0.88	0	0.31	0.27
morning	0.01	0	0.1	0	1	0.12	0.61	0.26
avg_60_forecastio_cloudCover	0	0	0	0.04	0.48	0.29	1	0.26
forecastio_humidity	0.28	0	0	0	0.55	0.26	0.69	0.25
derivative_avg_720_lmse_scaled_sharpDust	0	0	0	0.01	0.61	0.15	1	0.25
hour_of_day	0.02	0.41	0	0.01	0.22	0.01	1	0.24

Table 16: Top Features for Predicting AlphaSense CO Sensor 2

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
lmse_calib_as_co	1	0.1	0	1	0.18	0	1	0.47
evening	0	0	0.01	0.01	0.96	0.04	0.99	0.29
avg_60_forecastio_windSpeed	0.3	1	0	0.15	0.06	0	0.55	0.29
Solar Panel (V)	0.11	0	1	0	0.86	0	0	0.28
forecastio_windSpeed	0.27	0.62	0	0.02	0.29	0.01	0.72	0.28
sck_humidity_saturated	0.05	0	0	0	0.59	1	0.33	0.28
avg_15_as_no2	0.4	0.29	0	0.01	0.73	0	0.36	0.26
forecastio_fog	0.11	0	0.71	0	0.94	0	0	0.25
as_03	0.41	0.08	0	0.01	0.72	0	0.5	0.25
derivative_avg_1440_lmse_scaled_sharpDust	0.14	0	0	0.01	0.6	0.03	0.99	0.25
as_no2	0.41	0	0	0	0.71	0.01	0.53	0.24
lmse_as_no2	0.41	0	0	0	0.73	0	0.48	0.23
lmse_avg_15_as_no2	0.4	0	0.01	0.01	0.81	0	0.41	0.23
bkcarbon	0.05	0	0	0.13	0.5	0.1	0.81	0.23
avg_15_derivative_sck_temperature	0	0	0	0.01	0.57	0.41	0.53	0.22
daily_avg_forecastio_temperature	0.22	0	0	0.03	0.45	0.1	0.71	0.22
avg_15_lmse_as_no2	0.4	0	0.01	0.01	0.82	0	0.3	0.22
derivative_avg_360_lmse_as_no2	0	0	0	0.02	0.59	0.31	0.62	0.22
avg_60_bkcarbon	0.06	0	0	0.07	0.55	0.15	0.73	0.22
afternoon	0.13	0	0.01	0	0.97	0.06	0.31	0.21
avg_720_bkcarbon	0.02	0	0	0.18	0.53	0.06	0.69	0.21
alphaS2_work	0.02	0.61	0	0.03	0.23	0.02	0.49	0.2
avg_60_as_no2	0.39	0.01	0	0.02	0.99	0	0	0.2
avg_15_lmse_calib_as_co	0.94	0	0	0.05	0.05	0	0.35	0.2

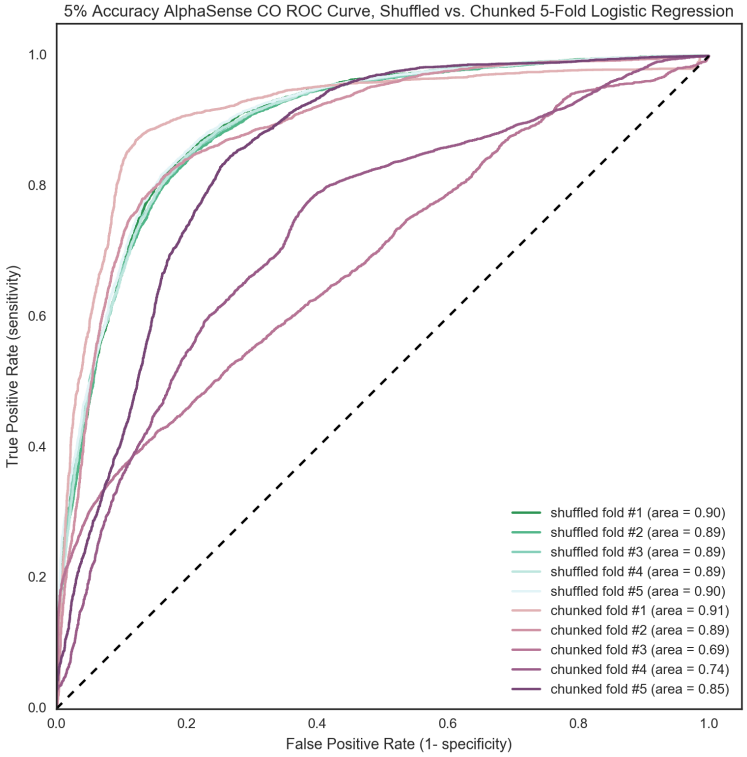


Figure 46: AlphaSense CO Sensor 1 ROC Curve

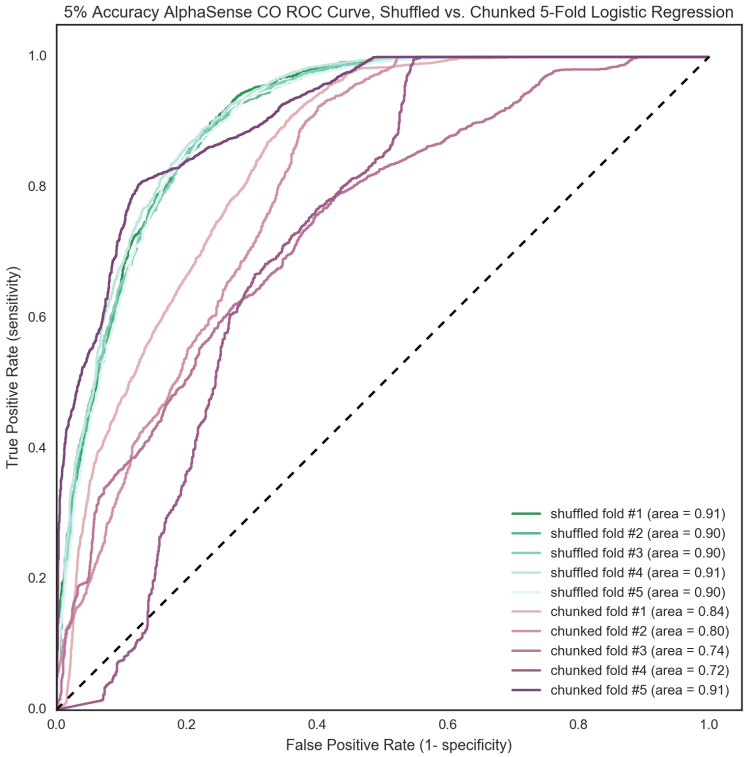


Figure 47: AlphaSense CO Sensor 2 ROC Curve

AlphaSense NO2

One AlphaSense NO2 sensor was tested against the EPA reference. It was 2 months old at the time of installation, and ran for 21 days (from 5/23 - 6/13 2016). This test gave 30,150 minute resolved samples.

Pre-processing

The AlphaSense NO2 raw data has similar concerns to its CO counterpart– it appears heavily 'quantized' based on the temperature regime it is in. The NO2 values were much further off than the CO values though, using the raw calculation.

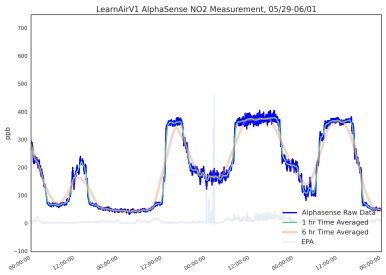


Figure 48: AlphaSense NO2 Raw Data Zoomed

Many techniques were used to tighten up the calibration after applying the initial datasheet equation and values, some of which greatly alter the variation, but at the end of the day a milder form was chosen based on scaling the electrode offsets. The final results can be seen in Figure 49.

Machine Learning

NO2 has some of the smallest variations of the pollutants we looked at. To assess its accuracy, 10% of its full scale value was used as a tolerance– this worked out to a very tight tolerance of ± 26.25 ppb. With this tolerance in place, our parameterized search of C = [0.001, 0.1, 10, 1000] and penalty-type=['L1', 'L2'] with a 2-fold cross-validation yielded an optima set of L1 penalty with C = 1000.

Even with this extremely tight tolerance, we can see in Table 17 that we have very low average error rates in all conditions, though our variability dramatically increases for the chunked cases. In conjunction with the ROC graphs of Figure 50 we can see that our seasonal predictive power is quite poor. This isn't unexpected, as this test (a three week test) was one of the shortest performed.

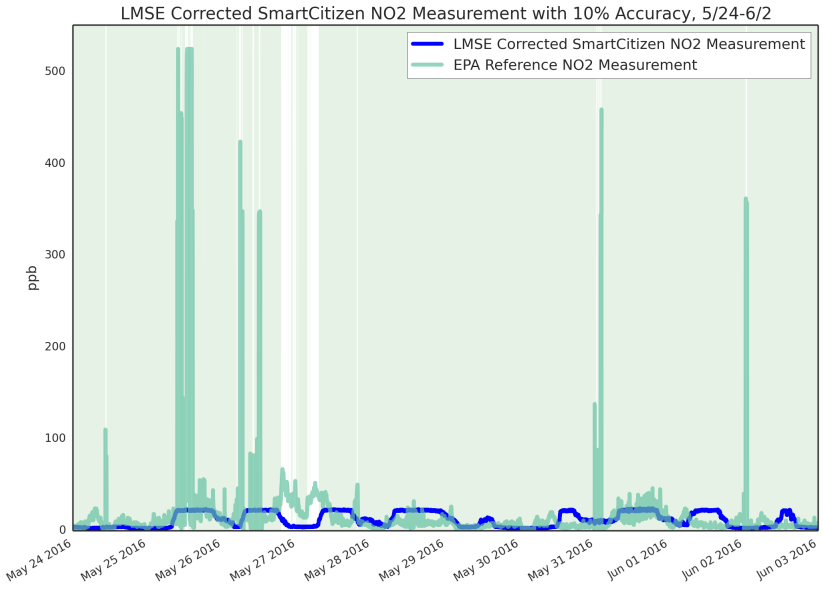


Figure 49: AlphaSense NO2 with 10% Accuracy Threshold

Error Rates for AlphaSense NO2 with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.03	0.06	0.04	0.04
min	0.03	0.05	0.03	0.01
max	0.03	0.14	0.04	0.14

Table 17: Error Rates for Predicting AlphaSense NO2 Accuracy with Logistic Regression

What is incredibly interesting here is that we see excellent predictive power with the shuffled case (AUC-ROC scores in the 0.96-0.97 range), contrasted with terrible scores for the chunked case (more frequently worse than random chance than not). As alluded to, this points to a lack of data for robust pattern-finding. The errors that co-occur in each period of time have uniquely defining features, but there are so few of them that no general pattern has emerged yet to define them all. The incredibly strong results in the shuffled case does not eliminate the possibility that there is a strong relationship to take advantage of here, but in general this test is inconclusive about the predictability of the NO2 sensor. The divergence of the shuffled and chunked cases strongly points to a need for more data before hard conclusions can be drawn.

The top features (Table 19) and the effect of reducing the feature set aren't particularly meaningful given our prior conclusion, but the fact that black carbon, cloud cover, and O3 concentration are in the

		Predicted Values	
		0	**1**
Actual Values	**0**	116.2	130.2
	1	34.0	5749.6

Table 18: Average AlphaSense NO2 Confusion Matrix w/Shuffled K-Fold

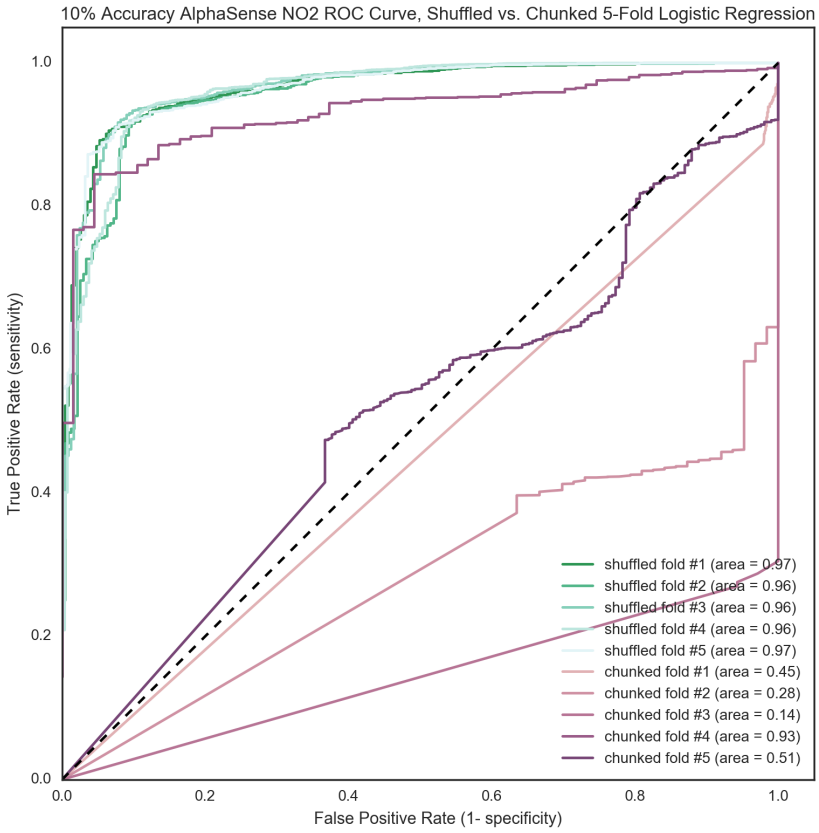


Figure 50: AlphaSense NO_2 ROC Curve

top several features (and these features seem to do a good job predicting robustness in the shuffled case by themselves) bolsters the hypothesis that there is likely a useful, predictable trend underlying this data. We would expect our high quality correlated sensor to provide a good backbone for prediction, and cloud cover and O_3 directly point to one of the most rapid and important reactions that converts NO_2 to other by-products. Since these highly relevant features drive our strong shuffled predictions (as oppose to spurious, coincidental ones), it seems likely that this AlphaSense NO_2 sensor will be highly predictable with a few more weeks of data collection.

Table 19: Top Features for Predicting AlphaSense NO2

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
bkcarbon	0.99	0	0	0.85	0.47	0.23	0.79	0.48
avg_60_bkcarbon	1	0	0	1	0.54	0.14	0.64	0.47
avg_1440_bkcarbon	0.9	0	0	0.16	0.5	0.44	0.69	0.38
daily_avg_sck_humidity	0.36	0	0	0.03	0.59	0.77	0.39	0.31
as_03	0.02	0	0	0.97	0.77	0.01	0.42	0.31
lmse_sck_co	0.06	1	0	0.01	0.81	0	0.28	0.31
avg_60_forecastio_cloudCover	0.16	0	0	0.11	0.53	0.38	0.81	0.28
Solar Panel (V)	0.05	0	1	0	0.87	0	0	0.27
derivative_avg_1440_bkcarbon	0.17	0	0	0.04	0.62	0.08	1	0.27
sck_humidity_saturated	0.04	0	0	0.01	0.58	1	0	0.23
avg_720_lmse_scaled_sharpDust	0	0	0	0.37	0.57	0.67	0.01	0.23
avg_720_bkcarbon	0.85	0	0	0.2	0.16	0.06	0.35	0.23
evening	0.05	0	0	0	0.95	0.04	0.47	0.22
day	0.06	0	0	0	0.99	0.06	0.42	0.22
night	0.06	0	0	0	1	0.06	0.41	0.22
alphaS2_work	0.31	0.78	0	0.02	0.21	0.05	0.09	0.21
forecastio_fog	0.05	0	0.4	0	0.93	0	0	0.2
forecastio_temperature_c	0	0	0.01	0	0.93	0.01	0.44	0.2
derivative_avg_720_bkcarbon	0.09	0	0	0.05	0.61	0.15	0.5	0.2
forecastio_temperature	0	0	0	0	0.92	0.01	0.41	0.19
Carbon Monxide (kOhm)	0.06	0	0	0.01	0.82	0	0.34	0.18
forecastio_cloudCover	0.19	0	0	0.01	0.44	0.17	0.48	0.18
forecastio_partly-cloudy-day	0.05	0	0.04	0	0.89	0.01	0.23	0.17

AlphaSense O3

Two AlphaSense O3 sensors were tested against the EPA reference. The first sensor was 2.5 years old at the time of installation, which ran for 38 days (from 4/15 to 5/23 2016 with one 40 minute service interruption). The second sensor was 2 months old at the time of installation, and ran for 21 days (from 5/23 - 6/13 2016). Our first test gave 55,589 minute-resolution samples to compare; our second test gave 30,150 samples.

Age is an important distinction between the two sensors, which makes this an interesting comparison. Additionally, while the first O3 sensor was only calibrated for O3 measurement, the second sensor was calibrated as an O3+NO2 sensor (to be used in conjunction with the NO2 sensor on the board). This presents a different calibration process- while the second sensor should be more accurate (as both are cross-sensitive to NO2), it depends on the accuracy of the NO2 characterization. The second sensor has a more complicated calibration equation, and thus introduces more opportunity for drift in the calibration process.

Pre-processing

The raw data for the O3 sensors is some of the most convincing we've recorded with a sensor at this price-point. The calibrated values for both sensors track the real values quite well, and even capture a few transients. A small sample of the datasheet-calibrated values is shown in Figure 51– for a look at all of the raw data, please check Appendix D.

The preprocessing step followed similar guiding principles as the other AlphaSense sensors. The final results for both sensors can be seen in Figure 93 (Sensor #1 on the left of the dotted line and Sensor #2 on the right).

Machine Learning

O3 varies the least of the pollutants we measured, and our 15% tolerance of full scale works out to be $\pm$15 ppb. This is extremely tight– the analog board is only rated for $\pm$10. The great thing is that a little

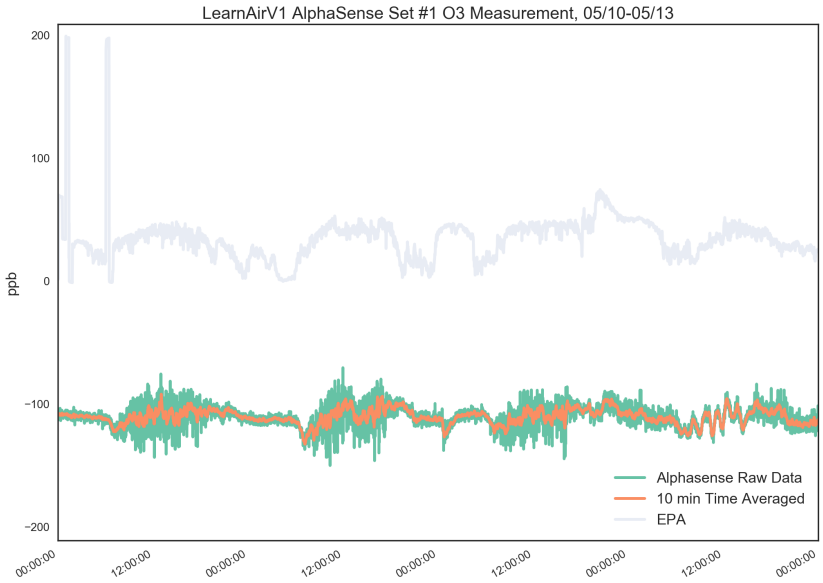


Figure 51: AlphaSense O3 Sensor 1 Raw Data Zoomed

less than half of our readings actually fall into that very strict tolerance. It may, however, be unrealistic to demand this level of precision from this sensor.

Our two fold search of the parameter space for both old and new O3 sensor gave an L1 penalty, and C values of 10 and 1000 respectively. As we can see from Tables 20 and 21, the errors levels we get when attempting to predict at this tolerance level are pretty extreme. The confusion matrix and ROC curves back up that assumption.

Error Rates for O3 Sensor 1 with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.33	0.43	0.37	0.41
min	0.32	0.37	0.36	0.36
max	0.33	0.52	0.37	0.52

Table 20: Error Rates for Predicting O3 Sensor 1 Accuracy with Logistic Regression

Error Rates for O3 Sensor 2 with Logistic Regression				
	all features		top 15 features	
	shuffled	chunked	shuffled	chunked
avg	0.26	0.46	0.32	0.40
min	0.24	0.37	0.32	0.35
max	0.26	0.54	0.33	0.49

Table 21: Error Rates for Predicting O3 Sensor 2 Accuracy with Logistic Regression

The newer sensor appears to be more predictable, even with less data– this may (again) be due to slightly more stable weather toward the end of the test. Both sensors have strong trends between shuffled

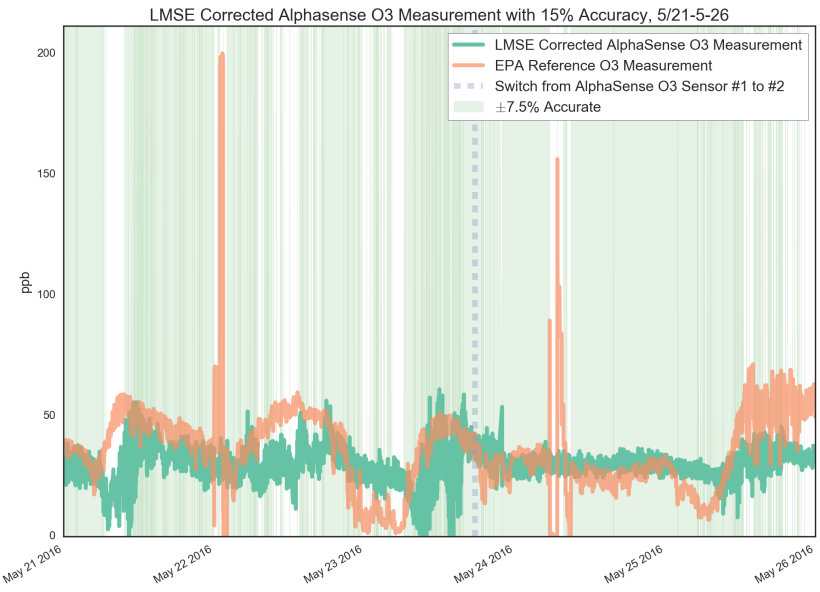


Figure 52: AlphaSense O3 Sensor 1 and 2 with 15% Accuracy Threshold

and chunked validation techniques, especially given the ambitious tolerance. They are also surprising resilient to feature set reduction.

As a next step, it would be useful to increase the tolerance to a more reasonable level and re-train our model for prediction. The quality of the readings suggests there is some very trustworthy and useful information to be had with this sensor.

		Predicted Values	
		0	**1**
Actual Values	**0**	4308.2	1730.2
	1	1931.8	3147.6

Table 22: AlphaSense O3 Sensor 1 Confusion Matrix w/Shuffled K-Fold

		Predicted Values	
		0	**1**
Actual Values	**0**	2439.6	747.4
	1	792.2	2050.8

Table 23: AlphaSense O3 Sensor 2 Confusion Matrix w/Shuffled K-Fold

Table 24: Top Features for Predicting AlphaSense O3 Sensor 1

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
alphaS1_work	0.32	1	0	1	0.44	0.14	0.94	0.55
as_h2s	0.55	0.84	0	0.08	0.96	0.01	0.63	0.44
avg_1440_lmse_scaled_sharpDust	0.22	0	0	0.08	0.52	1	1	0.4
avg_720_bkcarbon	1	0	0	0.24	0.54	0.27	0.59	0.38
bkcarbon	0.97	0	0	0.18	0.27	0.24	0.77	0.35
alphaS3_aux	0.68	0	0	0.04	0.97	0.02	0.67	0.34
forecastio_windSpeed	0.62	0.5	0	0.07	0.29	0.03	0.75	0.32
Solar Panel (V)	0.17	0	1	0	0.98	0	0	0.31
avg_10_as_03	0.19	0.35	0	0.05	0.51	0.29	0.76	0.31
Nitrogen Dioxide (kOhm)	0.59	0.04	0	0.04	0.74	0	0.67	0.3
lmse_sck_no2	0.59	0	0	0.04	0.74	0	0.73	0.3
avg_60_bkcarbon	0.84	0	0	0.19	0.49	0.02	0.59	0.3
derivative_avg_1440_bkcarbon	0.19	0	0	0.07	0.63	0.18	1	0.3
avg_1440_bkcarbon	0.78	0	0	0.25	0.53	0.36	0.01	0.28
derivative_avg_720_bkcarbon	0.08	0	0	0.05	0.61	0.21	1	0.28
daily_avg_forecastio_humidity	0.08	0	0	0.28	0.55	0.95	0.01	0.27
avg_15_derivative_avg_15_as_temperature	0.3	0	0	0.07	0.49	0.29	0.65	0.26
alphaS1_aux	0.01	0.93	0	0.06	0.45	0.13	0.17	0.25
avg_60_forecastio_temperature_c	0.42	0	0.01	0.07	0.76	0.01	0.48	0.25
humidity_box_differential	0.15	0	0.03	0.1	1	0.08	0.42	0.25
avg_10_lmse_calib_as_03	0.19	0	0	0.04	0.51	0.29	0.7	0.25
derivative_avg_60_bkcarbon	0.11	0	0	0.05	0.58	0.42	0.56	0.25
derivative_avg_1440_lmse_scaled_sharpDust	0.01	0	0	0.06	0.62	0.07	1	0.25
as_03	0.04	0	0	0.76	0.79	0.01	0.07	0.24

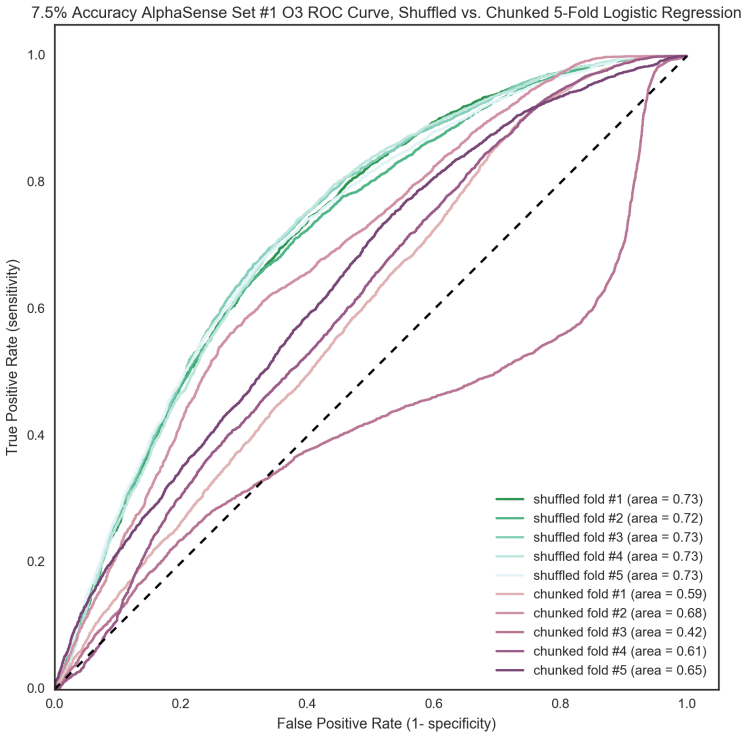


Figure 53: AlphaSense O3 Sensor 1 ROC Curve

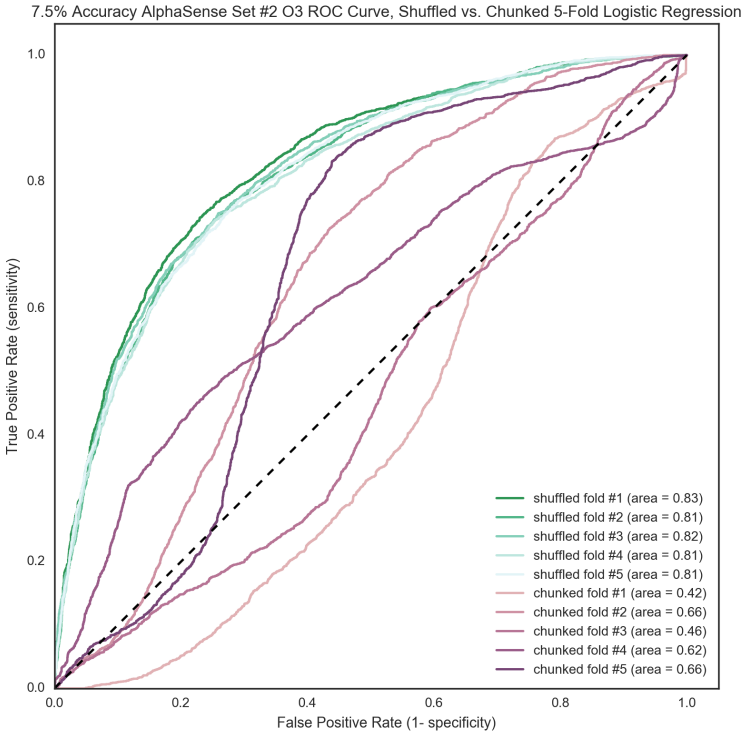


Figure 54: AlphaSense O3 Sensor 2 ROC Curve

Table 25: Top Features for Predicting AlphaSense O_3 Sensor 2

	Corr.	Lasso	Lin Reg	RF	RFE	Ridge	Stability	Mean
avg_1440_as_co	1	0.12	0	0.58	0.08	0	1	0.4
daily_avg_sck_humidity	0.71	0	0	0.07	0.57	1	0.36	0.39
avg_1440_bkcarbon	0.66	0	0	1	0.48	0.33	0.01	0.35
avg_60_bkcarbon	0.79	0	0	0.38	0.35	0.06	0.8	0.34
forecastio_pressure	0.27	0.82	0	0.1	0.33	0.04	0.75	0.33
avg_60_forecastio_apparentTemperature	0.17	1	0	0.07	0.4	0.09	0.57	0.33
avg_15_derivative_sck_temperature	0.03	0	0	0.04	0.57	0.45	1	0.3
bkcarbon	0.73	0	0	0.14	0.47	0.04	0.66	0.29
Solar Panel (V)	0.1	0	1	0	0.86	0	0	0.28
avg_720_bkcarbon	0.76	0	0	0.38	0.45	0.23	0.12	0.28
derivative_avg_1440_lmse_calib_as_co	0.16	0	0	0.08	0.51	0.14	1	0.27
daily_avg_as_temperature	0.79	0	0	0.19	0.43	0.11	0.31	0.26
lmse_avg_30_scaled_arduino_ws	0.34	0.06	0	0.04	0.94	0.01	0.4	0.26
avg_1440_lmse_scaled_sharpDust	0.05	0	0	0.1	0.53	0.39	0.77	0.26
derivative_avg_720_lmse_scaled_sharpDust	0.1	0	0	0.08	0.61	0.05	1	0.26
alphaS1_aux	0	0	0.03	0.04	0.69	0.02	1	0.25
avg_30_scaled_arduino_ws	0.34	0	0.01	0.04	0.95	0	0.41	0.25
derivative_avg_1440_bkcarbon	0.09	0	0	0.06	0.61	0.01	1	0.25
forecastio_cloudCover	0.23	0	0	0.16	0.49	0.01	0.64	0.22
forecastio_clear-day	0.01	0	0.03	0	0.89	0.02	0.56	0.22
avg_60_forecastio_pressure	0.28	0	0	0.2	0.33	0.03	0.72	0.22
Nitrogen Dioxide (kOhm)	0.1	0.13	0	0.04	0.78	0	0.41	0.21
forecastio_temperature_c	0.17	0	0.01	0.27	0.87	0.05	0.06	0.2
avg_60_as_no2	0.03	0.04	0	0.09	0.77	0	0.47	0.2

Results Summary

Logistic Regression was used to predict the accuracy of six types of sensors. The primary metrics for success were the AUC-ROC values for the shuffled and chunked cross validation sets, the error rates from our confusion matrices, and the believability/consistency of the top predictive features after feature reduction. The AUC-ROC is the only metric that takes into account our prediction of the likelihood that our classification of each reading (and thus AUC-ROC is the strongest indicator of algorithmic success).

After working with the data, we found a few important trends. (1) Using both shuffled and chunked cross-validation techniques, we found that divergence in their results suggests that we haven't collected enough data to account for time-variant, seasonal effects. (2) We found that calibration was a difficult step, especially when considering sensors that clearly work some of the time, and are largely in error at others. LMSE might not be the best solution in this case, because it assigns extra weight to large errors (when it would be most beneficial to our calibration if we ignored them). While we tried several techniques to achieve get around this (mean absolute error, ignoring large errors, throwing out outlier values and doing a LMSE fit on just the values close to the average), this stage of data processing is heavily manual, requires intuition, and can break in unexpected ways. (3) We saw our sensors missing high-concentration transient events, though the one or two they did capture demonstrate the ability of the sensors themselves to measure such occurrences. This is likely an airflow problem, and something important to address in the future. Finally, (4) we can ascertain interesting insights about sensor quality and self-noise (as oppose to predictable, systematic failures) by examining the relationship between tolerance and the machine learning success. These tolerances were chosen empirically in this thesis, but this step could be automated by training models for several tolerances and exploiting inflection points in the prediction results.

Besides these insights about the machine learning process, we were able to predict the accuracy of our sensors with success. All of the results are summarized in table 26, and discussed briefly below.

We noticed that the Smart Citizen sensor provided little useful information at these pollution levels, and prediction in this case reduced to a different problem– the prediction of transient events (instead of

the prediction of sensor accuracy). The results of this analysis show that we are reasonably able to predict transients, especially NO2 transients. In both cases, seasonal variation appears to have a slight effect on the quality of our results. Both CO and NO2 transient prediction rely on black carbon measurements as their main feature– the only reference grade sensor used as a training value. This fits our expectations (since all three result from combustion), and the commonality gives us confidence that this is a real phenomena. This result suggests that one high quality sensor could ground strong data quality predictions for cheaper sensors measuring related pollutants.

The raw sharp data shows that it closely tracks the federal reference most of the time, but for some periods it diverges greatly. This behavior is suggestive of a systematic failure (and thus promising for our techniques). Unfortunately, this is our smallest dataset, because reference data is only available on an hourly basis. Our results show that we need more data for predicting across seasons. Despite this, we see incredibly strong predictions in the shuffled case, especially for 48 hour averages. These data are very promising.

The AlphaSense CO sensors clearly demonstrate responsiveness to fine changes in pollution concentration. While the detail matches our reference data, there are still issues with the relatively coarse temperature compensation and calibration. It would be beneficial to explore new methods for temperature-based calibration techniques.

Regardless of the pre-processing, we achieved strong predictive results for both AlphaSense CO sensors tested. Both gave similar results, and both had their own signal as their top feature (indicating that the AlphaSense CO sensor has data ranges that were very reliable as well as ranges that were more likely to be inaccurate). Based on the shared features between the two sensors tested, it appears that wind and temperature likely factor into the CO predictions. There is some discrepancy as to whether black carbon and NO2 are strong predictors, so more data would be useful.

The AlphaSense NO2 results were unique. We see low error rates and very strong predictions when shuffling the data, but the results are highly variable when not shuffled. It does not appear to be a random relationship, either– the shuffled results can be very strong (AUC-ROC of .93) or very poor (.14, .28). This suggests that there are useful predictive trends that vary drastically with the seasons. (For example, humidity may be a strong predictor in the cold months, but not in warm ones. If you try to apply the principles learned in the

Summary of Results				
	avg prediction error		avg AUC-ROC	
	shuffled	chunked	shuffled	chunked
SmartCitizen CO	0.09	0.09	0.82	0.61
SmartCitizen NO2	0.03	0.03	0.90	0.77
Sharp Dust	0.17	0.22	0.87	0.79
48hr avg Sharp Dust	0.08	0.25	0.98	0.79
AlphaSense CO	0.16	0.21	0.90	0.81
AlphaSense NO2	0.03	0.06	0.96	0.46
AlphaSense O3	0.30	0.45	0.77	0.58

Table 26: Summary of Machine Learning Results

cold as it gets warm, you will confidently assert incorrect predictions and do much worse than guessing.) Despite the variability in the chunked data, we see strong results in the shuffled data that imply success across the seasons with more data collection.

Finally, the AlphaSense O3 raw data (like CO) tracked small variations in the reference grade equipment quite well. Like the CO and NO2 sensors, the main issue in the quality of the results was the lack of captured transients. In this test, we saw slight differences in results from the newer and older sensors. The newer sensor provided better results, with AUC-ROC scores averaging 0.81 versus 0.73, though this is confounded by the changing weather.

We used the tightest tolerance for the AlphaSense O3 sensor ($\pm$15 ppb when the front end board is only spec'ed for $\pm$10 ppb), which is why the results aren't the strongest of the set. They are still quite good considering this decision. Loosening this tolerance would likely improve predictive behavior. More data is required to make strong O3 predictions across seasons.

8. Conclusions and Future Work

Insights

Instead of endeavoring to build a *better* inexpensive sensor, in this work we attempted to *understand* and *predict* the sensors we already had. These techniques have wide implications for the heterogeneous, dense, and dynamic sensing ecosystem of the future.

We tested a machine learning approach to this problem on six different sensor types, and found strong predictive results in some cases. In other cases, we discovered hopeful results, so long as the sensors provided useful information. The cheapest sensors– the Smart Citizen CO and NO_2 sensors– did not provide any useful data. The Sharp particulate sensor clearly followed trends with some large divergences– it showed very strong predictability with a reasonably large tolerance. The AlphaSense sensors exhibited interesting behavior– in every case, they moved from one temperature-dependent regime to another throughout the day. This switch in temperature overlays a large step behavior on top of the measured signal which otherwise appears to track real pollution concentration. This extent of this temperature dependent 'quantization' has an adverse effect on data quality for the concentrations we observed.

Though strong predictability was observed in several cases, the inexpensive sensors failed to record most transient phenomena. This trend is likely related to airflow around the device. For CO and O_3, a few small transients were recorded by our sensors. A comparison of transient duration and the sensor time constants also supports the hypothesis that these sensors can detect such events. The NO_2 transients were much faster, and the underlying problem is harder to dissect. New designs that promote airflow should be built to test, understand, and mitigate this issue in the future.

Contrasting 'chunked' and 'shuffled' cross-validation emerged in this work as an interesting technique. The differences between the two helped us understand whether we had collected enough data to predict future or past data reliability given seasonal variation. When these results align, it is a strong indication that we have captured enough data to make meaningful predictions across measured seasons. This test's two month window limited our capacity to fully characterize and predict the sensor reliability across season. This is unsurprising given the dynamic weather and the short, early exposure to cold. In some instances the chunked cases show one or two poorly predicted major outliers. These may correspond to the majority of temperate weather data training a model to predict the coldest week at the beginning of the test, or generally dry and clear weather sections predicting the rainiest and most humid week. By applying this cross-validation strategy, we provide an automatic, quantifiable, and objective measure of the dataset integrity.

Furthermore, the machine learning can be used as a tool to quantify sensor quality. The predictability of failure is a strong indicator of the quality of the device's underlying physics and design. Additionally, by varying the threshold of what we consider an 'accurate' reading and seeing how the quality of our model's predictions change, we can tease out the precision of a device. We expect a steep decline in predictive power of our model when we cross the threshold from systematic, predictable errors to the inherent resolution limit of the device. By characterizing this break-point, we can make objective claims about a device's precision.

Feature reduction gives us an idea of the failure modes and/or correlates for each sensor. In our tests, the top features were mostly in line with expectations, which is a strong indicator that the results we have observed are meaningful. For instance, the most relevant Sharp particulate sensor features suggested it has (1) an operating range for which it works well, and one for which it is less reliable, (2) covaries with other pollutants, especially NO_2, and (3) is sensitive to rain and humidity. Clusters of similar features arising from different feature reduction algorithms corroborate these conclusions.

Interestingly, the black carbon sensor reading– one of the only readings from the EPA reference set to appear as a training feature– was revealed to be the strongest predictor in nearly every case in which it was included. This provides fundamental insight into system design– one, high quality sensor on a device may be invaluable at predicting and validating cheaper sensors on that same device. We can use these

machine learning techniques to test possible combinations of expensive and cheap sensors, and thus optimize and inform our system design.

By selectively removing these accurate data points taken by an expensive sensor, we move away from using our tool (1) to understand the underlying mechanisms of how the device breaks down and (2) as a method to inform future system design. Instead, we can use it to characterize a pre-existing system, and explore whether we can extract more accurate results simply by using multiple low-cost sensors. The applications of these machine learning techniques for sensor characterization, system characterization, system design, data analysis, and data quality assurance are numerous.

Beyond the machine learning and data itself, there are other important insights to consider from this work. In this thesis, we built multiple hardware platforms. In the first, we found airflow to be a likely issue for directional selectivity and sensitivity of the device. We tested an inexpensive, custom wind sensor using differential pressure techniques and showed both the complexity and the promise of such a design. For our purposes it was a useful proxy for local airflow, but there are many open questions and future applications for this wind sensing modality.

Finally, there are insights regarding the backend design of the learnAir system itself. The implementation exposed some weakness in the HAL/JSON standard and ChainAPI implementation for these applications– namely, (1) that resources do not have a self-description of their type (i.e., they are defined through link relationships, so you must maintain state as you traverse to know what type of resource you have), (2) that plural and singular forms are implemented when grouping like resources as lists, which can exacerbate confusion when traversing, querying, working with code, and defining ontologies, and (3) data storage as it is implemented in ChainAPI serves data in small date-range page chunks, with no clear start or end date (i.e., it is currently impossible to distinguish a large break in data from the beginning or end of data collection without external context).

Although these issues warrant attention, none are indictments of the system as a whole. In fact, ChainAPI addresses many important concerns facing the air quality community, while also providing provocative benefits over standard database solutions. ChainAPI lowers the barrier to entry, allowing for simple distributed hosting, sharing, and

ontology curation. Beyond that, it enables separate concerns so that raw data is transparent, data processing can be centralized, and best practices can emerge in an open ecosystem. This represents an important paradigm shift away from opaque self-curation and publishing of data towards a world where data is automatically uploaded, calibrated, and processed directly from hardware. With this thesis, that platform has been expanded using tools to take advantage of ChainAPI as an easily discoverable, scalable, dynamic, and browsable system (a la the world wide web). There are many important architectural insights inherent in this structure– pushing away from the standard static monolithic solutions, and towards a dynamic, open system.

Applications

The applications for this work are numerous. The algorithms themselves show promise for quantifying and standardizing many decisions in the real-world– they can be used to rigorously and automatically characterize sensor failure modes and sensor quality. They can objectively show us when we have enough data to understand how a sensor reacts in a given climate. When paired with a reasonable quality sensor, they can give a probability of data accuracy for each reading, priming researchers to use and explore new, large, distributed datasets and novel air quality models that factor in data uncertainty.

The data backend has many applications for the issues facing air quality sensor networks. The air quality community has been debating approaches to share data, define a common ontology, and engage groups with lower quality sensor distributions. With the latest implementation, ChainAPI has been adapted and expanded to support the needs of this diverse, dynamic air quality ecosystem. It enables new modes of interaction with large datasets beyond the basic needs of air quality, and points to architectural advances for large scale distributed sensor networks in general.

Beyond the obvious architectural features, the tools for ChainAPI represent one of the first scalable semantic web toolkits. It allows basic discovery, collection, and manipulation of data in a dynamic, web-based data structure. It provides a framework for learning models that automatically find relevant data to update their state, as well as crawl through the web and process/publish data automatically.

This is extensible to any type of algorithm, not only machine learning ones.

There are many potential applications for such a system. The obvious example is learnAir itself– a network of variable quality sensors that automatically learn from their neighbors, as well as similar distant sensors of a similar make and model. From this network, it would be possible to blend the data and create a highly resolved and accurate pollution map. Other useful applications are not hard to find, though. This system could also enable a simple tool that automatically characterizes and ranks new consumer devices– similar to the process SCAQMD and EPA are doing now. This would save manual labor, provide a more nuanced picture of the sensor in different climates and conditions (i.e., instead of a simple overall ranking), and standardize test procedures. Furthermore, as consumer devices connect to ChainAPI, it is simple to build a map that uses the latest data to display the best sensors for each location, based on its climate. Sensors could be certified for regions, conditions, or climates, all using real-world test data.

While the backend is ripe for simple tasks like automatic calibration, learning algorithms are still at a stage where it is important for a human to be in the loop. It is important to check outputs and feature relationships against sophisticated intuition of the device physics to ensure reliability. In the future, this could change with meta-analysis and classification of common modes of failure (i.e., teaching an algorithm to recognize and validate common failure modes such as cross-sensitivity errors or commonly co-variant pollutants, etc.). This second level machine learning task is an interesting future direction to explore in the context of ChainAPI.

Overall, the backend system for learnAir provides interesting, novel ways to explore large-scale ecosystems with variable quality data. ChainAPI and the new ChainAPI tools demonstrate a unique and compelling vision for the future of large scale, distributed, dynamic data storage and interaction systems where data quality varies and data processing is complex and decentralized.

Finally, the learnAir hardware represents a step forward for data assurance in the cheap, mobile space. The algorithms we have used can intelligently inform hardware system design– for example, which sensors work best together to create a trustworthy system. It is likely, for instance, that one slightly nicer sensor can significantly improve the reliability estimates for cheaper sensors that measure correlated

pollutants. Compared with similarly priced and designed systems, learnAir has better data quality. Although obstacles remain in translating the outcomes of this work from a static context to a mobile one, we have demonstrated in this thesis that the core principles are sound. A device of this type can empower users to improve their health with data that they can trust. It also provokes a conversation in the citizen sensing community– inherent to the design are questions of data quality and complex sensor modalities. LearnAir could play a small role in educating the population about underappreciated challenges with air quality sensing.

Future Work

LearnAir begets as many questions as it answers. There are many steps to take in the future to advance this work.

Questions pertaining to core sensor technology and data quality remain unanswered. For example: is there a way to promote the capture of the pollution transients we missed? This requires more co-location tests, new device geometries, and new airflow systems, since we hypothesize airflow is largely to blame. Additionally, we discovered that two months was not an optimal length of time to collect data capable of strong cross-seasonal predictions. It would be useful to run an extended co-location test, with additional sensors.

Airflow measurement is extremely important for the designs. While we showed promise using our differential pressure design, a great deal of work remains if we intend to optimize and/or orthogonalize this cheap wind sensor and corroborate its accuracy.

Taking this work from a stationary context to a mobile one is also an important goal. It may be possible to take this test on the road by mounting one of our sensors to a trustworthy mobile reference, like one of the Google-Aclima cars. Our partnership with EDF suggests this is not unreasonable. It would also be possible to purchase or rent high quality (>$15k) portable sensors as a reference, and walk or bike with both. It would also be possible to spin or shake one of our sensors around a fixed sensor at different speeds with a test rig to simulate motion and airflow at different walking and biking rates.

Besides the questions of data collection and hardware design, ChainAPI could use more tools and visualizations for air quality data. In the

short term, many of these machine learning scripts take hours to run on a laptop (and this is a reduced test, with just one sensor and two months of data). We are implementing these algorithms on Amazon Web Services cloud clusters, and it would be very useful to tie in these massively parallelized resources to ChainAPI with simple tools. Other next steps include designing ways to handle and integrate laboratory test data in ChainAPI, building out examples of calibration and manufacturer scripts (to check sensor data for the spec'ed operating range or service schedule), and to build out more tools and algorithms to simplify interaction with the data. The most important next steps around ChainAPI involve socializing it in the air quality community and soliciting feedback about the ontology and usability of the system.

On top of the ChainAPI backend, a few new websites or services could be created. We believe this infrastructure could help automate device testing for large air quality organizations, and drive a data-rich display of tested devices. Using this information, it would be simple to build out a user-friendly recommendation system for devices that perform well in different environments. It would also be trivial to apply this logic to the sensor level (looking at cost and performance under various conditions) in addition to the device level.

There are many potential next steps and explorations to engage in with machine learning itself. We started with a straightforward approach to error– a binary classification task. Regression analysis provides the ability to predict the magnitude of the error instead of just the existence of one. This could move us closer to true improvement in the data quality instead of just quality control. Additionally, deep-learning, time-dependent techniques could provide important insights into the effect of time variant phenomena. Much of the machine learning landscape is still left to explore. We have built our tools using scikit learn (a python library), and started basic exploration of tools such as Weka and Google's Tensor Flow– it would be an interesting next step to build that into our ChainAPI infrastructure. Additionally, there are open questions about the best way to approach calibration and pre-processing, as well as its automation. While we attempted several algorithms, we have not arrived at a strong solution; and this part of the process was more laborious and unintuitive than the actual application of machine learning. There is promise for intelligent, automatic techniques, but it would require an intense research effort to bring it to fruition.

Additionally, our API data is not yet fully inclusive. Pollen level,

for instance, is extremely important for particulate sensing, though no public pollen API exists at the time of this writing. There are opportunities to scrape this data, however. While things like pollen are obvious, the sophistication of publicly available derived features can easily grow. Real time traffic data, traffic type (percentage heavy diesel) based on road type, location, and time of day, construction information, road age/type/condition, distance to the nearest street, nearby building density and height, etc., could all be inferred from public map data and GPS coordinates. These features could have a profound effect on predictive accuracy.

At the end of all of this is a very fundamental question, which we have not examined in this thesis– how useful is data with a probabilistic quality indication? Certainly for individuals, it will give a measure of trustworthiness and a better personal prediction. It can even pull and plot trustworthy data from the closest reference sensor when it detects an unreliable measurement. For researchers, however, this type of dataset requires using new techniques to deal with datasets probabilistically, instead of the current approach which assumes all data is of high quality. It is important to explore the types of analyses and conclusions this work might enable, as we move towards more advanced statistical analyses, based on larger, distributed datasets.

Finally, a large part of this work deals with community engagement and interaction. In the short term, this means writing articles and journal papers on this topic, engaging community organizations, and advancing the dialogue around sensor quality using learnAir as an anchor point. Ultimately, as the pieces of this project are further refined, we would like to build a production-quality, mobile device that is trustworthy and useful to individuals, as well as an ecosystem where they can engage with and ground the air quality data they have collected.

Summary

The goal of this thesis was to create an affordable device that knows when it is within a given radius of a higher quality reference, compares itself to that reference, and learns when it is reliable and when it is not based on the context of its measurement. To build such a system, we needed to (1) test machine learning algorithms to ensure the fundamental approach would work, (2) build an infrastructure for

this device to post data, find nearby sensors, and automatically update a machine learning model of itself, and (3) create the hardware that captures contextual data about a measurement and talks to that infrastructure through a GPS enabled phone application.

We succeeded to build out all of these pieces. We built hardware with six affordable sensors, and ran a two-month co-location test to verify that machine learning could be used to predict sensor error. Results were mixed based on the sensor, but in general there is a lot of promise for these techniques to provide meaningful context to a sensor reading. We also found that machine learning could be used to draw conclusions about sensor quality and seasonal variation.

We built a database structure that can automatically apply these algorithms to sensors in the network and learn as more devices are added. This structure may also serve as a valuable tool for general issues of data sharing in the air quality community.

Finally, we built a provocative piece of mobile hardware that uses these core principles. There is still work to verify these principles in a mobile context– however, this hardware should serve as a simple way to test those principles, as well as a provocative tool to engage citizen groups in a meaningful dialogue around sensor quality. It is our hope and belief that in the near future a full characterization of the mobile system– together with data leveraged from other, nearby sensors– will enable the highest quality, truly mobile sensor system on the market.

Appendix A - Notes on Project Selection and Prior Work

Without a background in air quality before this year, a large portion of the thesis was getting up to speed with the community, learning where the issues are, and trying to understand where advances might be made. The thesis you've read is drastically different than the thesis I proposed ten months ago.

The first path we started down was to design and build some basic prototypes for chambering air samples. The concept was to introduce these devices in high wind environments (like on a car or bicycle), and sample the air by sealing it in a chamber and tagging it with GPS coordinates. Once a reading from an electrochemical gas sensor or another type of sensor had reached steady-state, we would evacuate the chamber and resample. This inter-sample settling time could be learned and/or evaluated in real-time for varying conditions.

Below are mockups of two prototypes we 3D printed and prototyped for AlphaSense electrochemical gas sensors. One is modeled after their calibration accessory (with a motorized paddle that seals the chamber when closed), and one is modeled after a revolver (with a spinning opening). Another prototype we discussed was a syringe style design, which has benefits for controlling chamber pressurization.

The first of these was physically prototyped using the AlphaSense AFE board, an Arduino, and a ReadBearLabs Bluetooth LE Arduino Shield. It was connected to a smartphone where it reported values, and could be commanded to open or close the chamber from the phone.

After several more discussions, I decided to change directions and focus on some of the problems that I perceived to have more direct impact. In general, there seems to be trust of the AlphaSense sen-

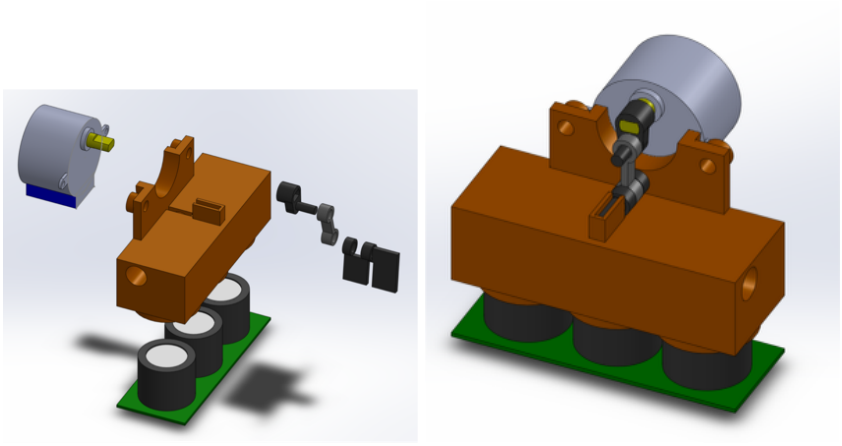


Figure 55: Original Concept #1

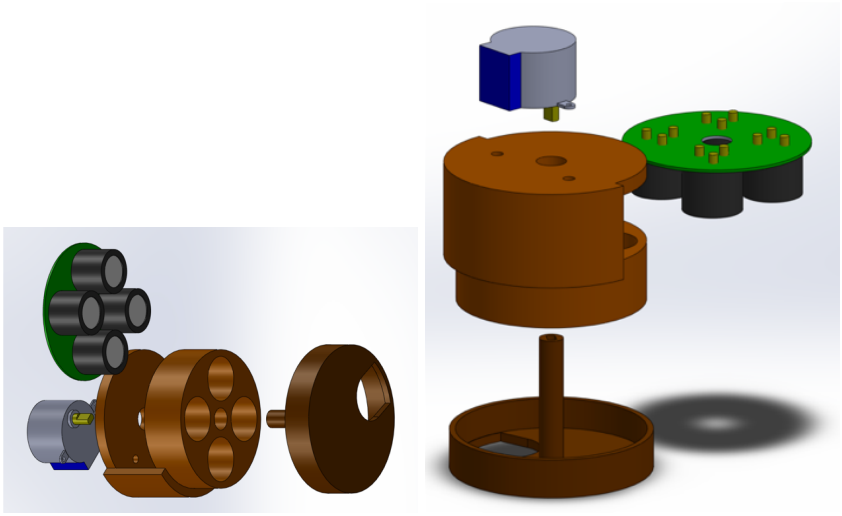


Figure 56: Original Concept #2

sors in a mobile context, without discrete air sampling (regardless of whether this is a correct assumption– to my knowledge, it has not yet been tested appropriately). The needs of the air quality community seem to be at an earlier stage– how to share data, how to characterize consumer devices, and how to understand consumer sensor quality. It seemed to be a waste of time to build another consumer device in a crowded space, when there are no standard mechanisms to verify its quality or spread its impact in the air quality community. Additionally, there is skepticism towards mobile-first sensing. There are still issues facing affordable stationary sensing, and no rigorous precedent on which to validate truly mobile solutions.

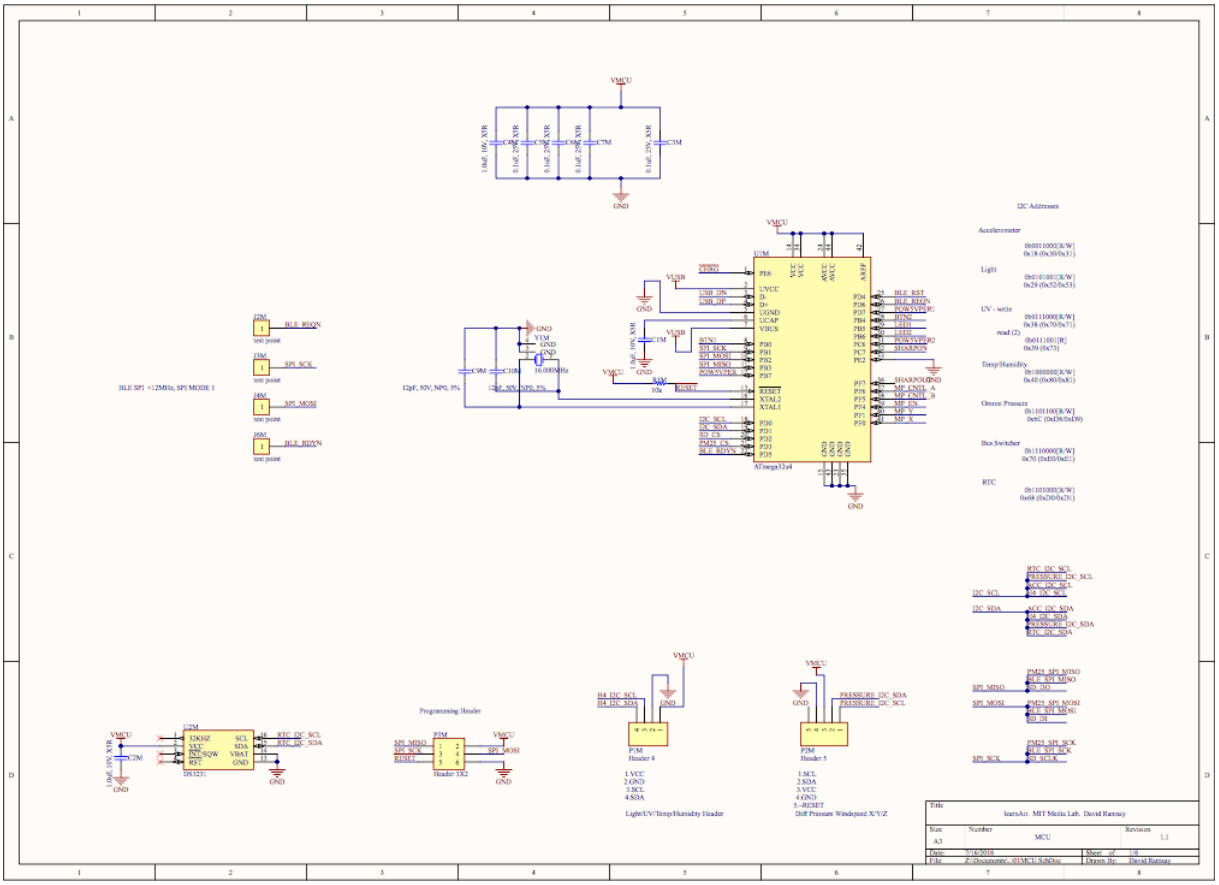
It is my hope that within a few years, the leaders in the air quality community will be positioned to make strong claims about whether

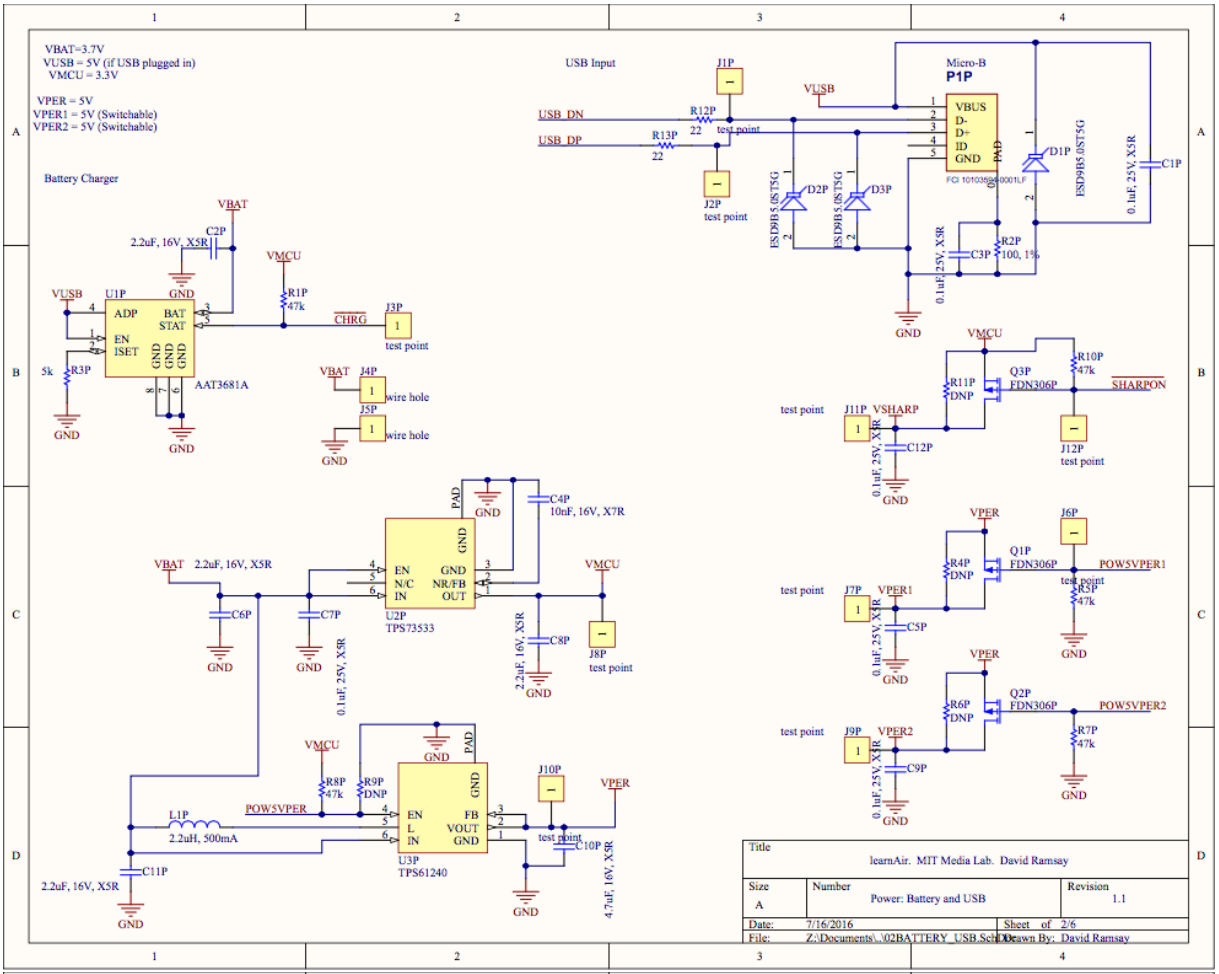
new devices (like the one proposed above) actually work. I hope this work may help form the foundation for which those claims can be made. While the goal is mobile, personal sensing, the principles herein apply equally strongly to affordable, dense, heterogeneous, sensor systems, which certainly will feature prominently in the future of air quality measurement.

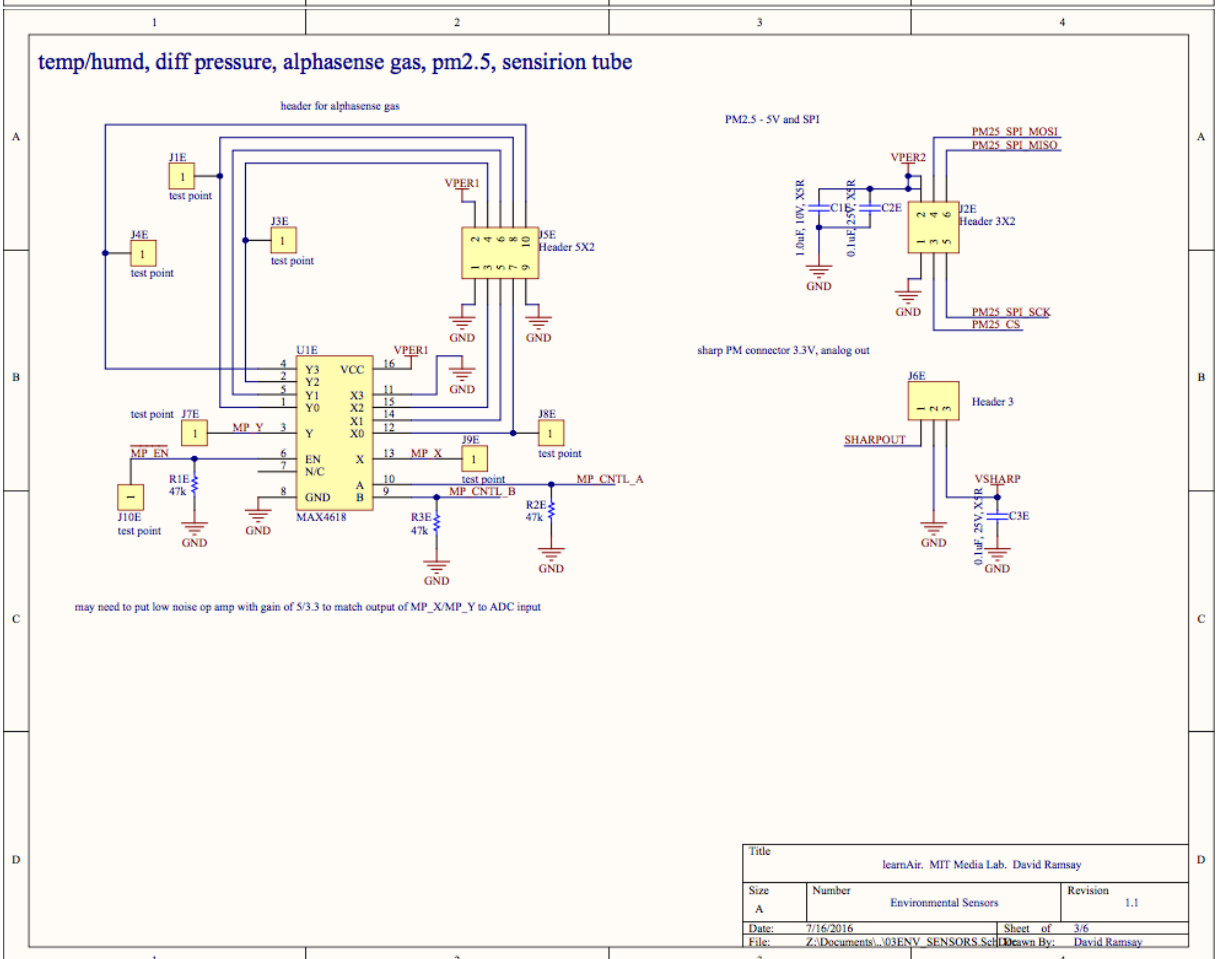
Appendix B - Hardware and Firmware

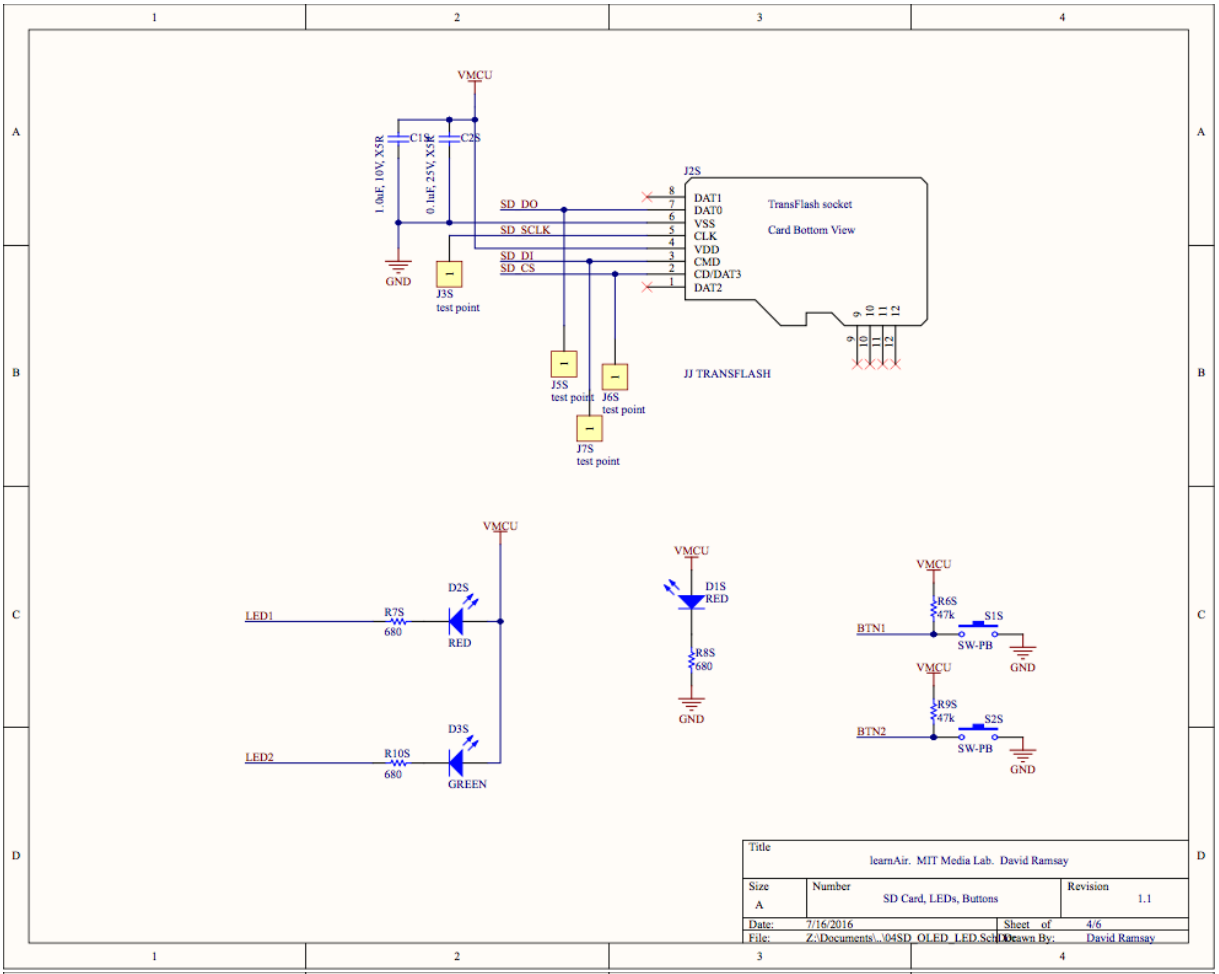
Schematics

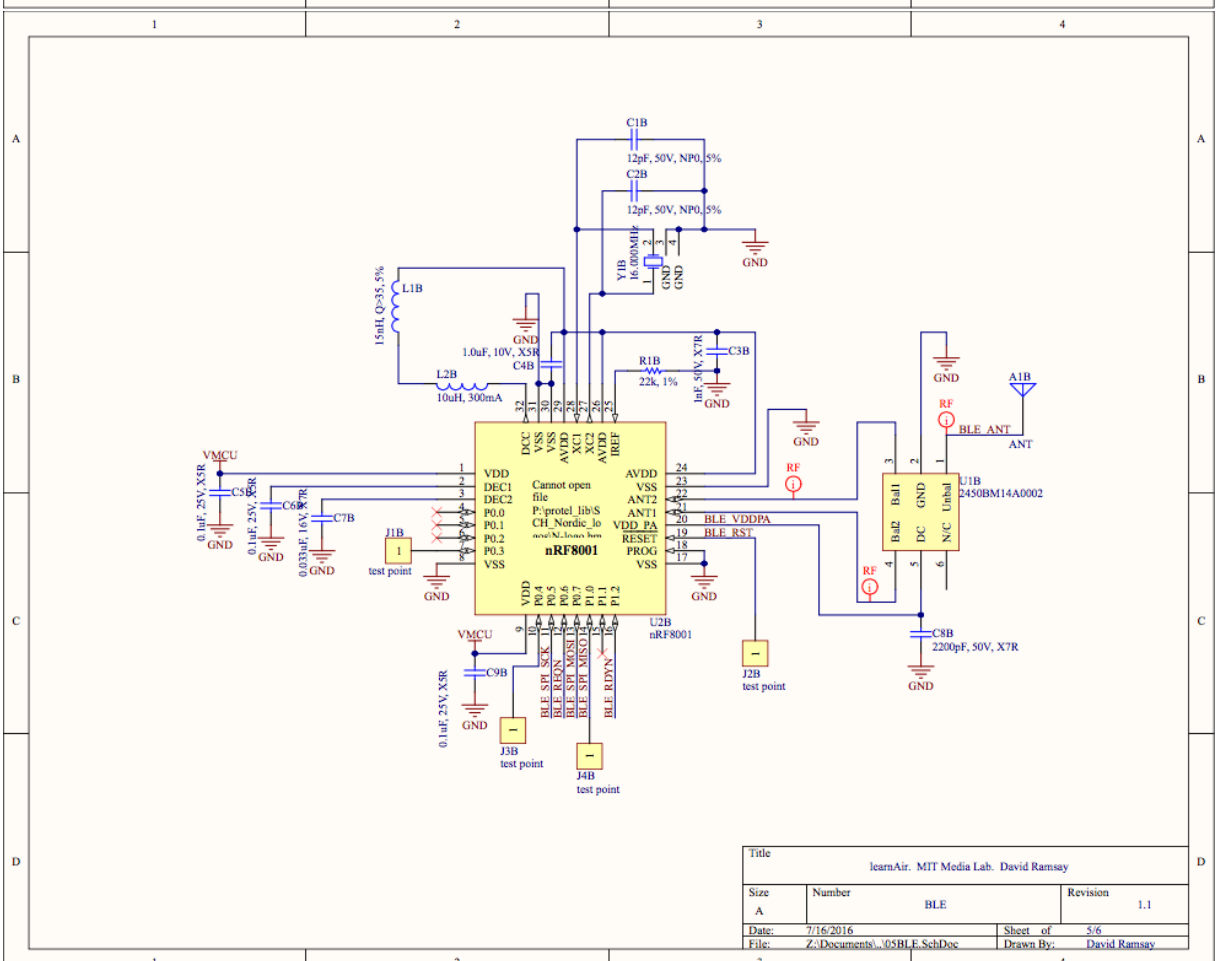
LearnAir V2 schematics are below.

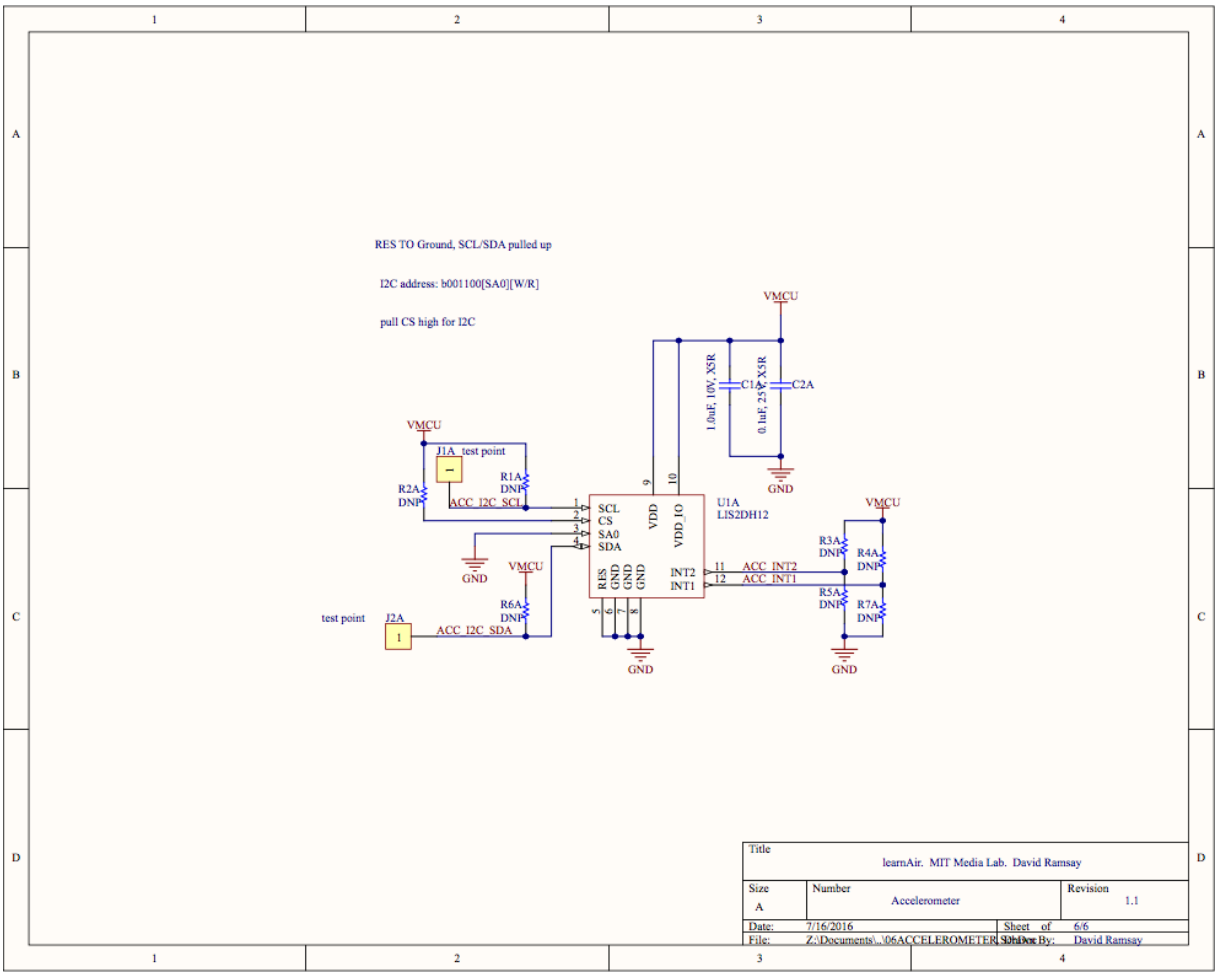



VBAT=3.7V
VUSB = 5V (if USB plugged in)
VMCU = 3.3V
VPER = 5V
VPER1 = 5V (Switchable)
VPER2 = 5V (Switchable)
Battery Charger
USB Input
Micro-B
P1P
AAT3681A
TPS73533
TPS61240
FDN306P
learnAir. MIT Media Lab. David Ramsay
Power: Battery and USB
Revision 1.1
Date: 7/16/2016
Sheet of 2/6
File: Z:\Documents\..\02BATTERY_USB.SchDoc
Drawn By: David Ramsay

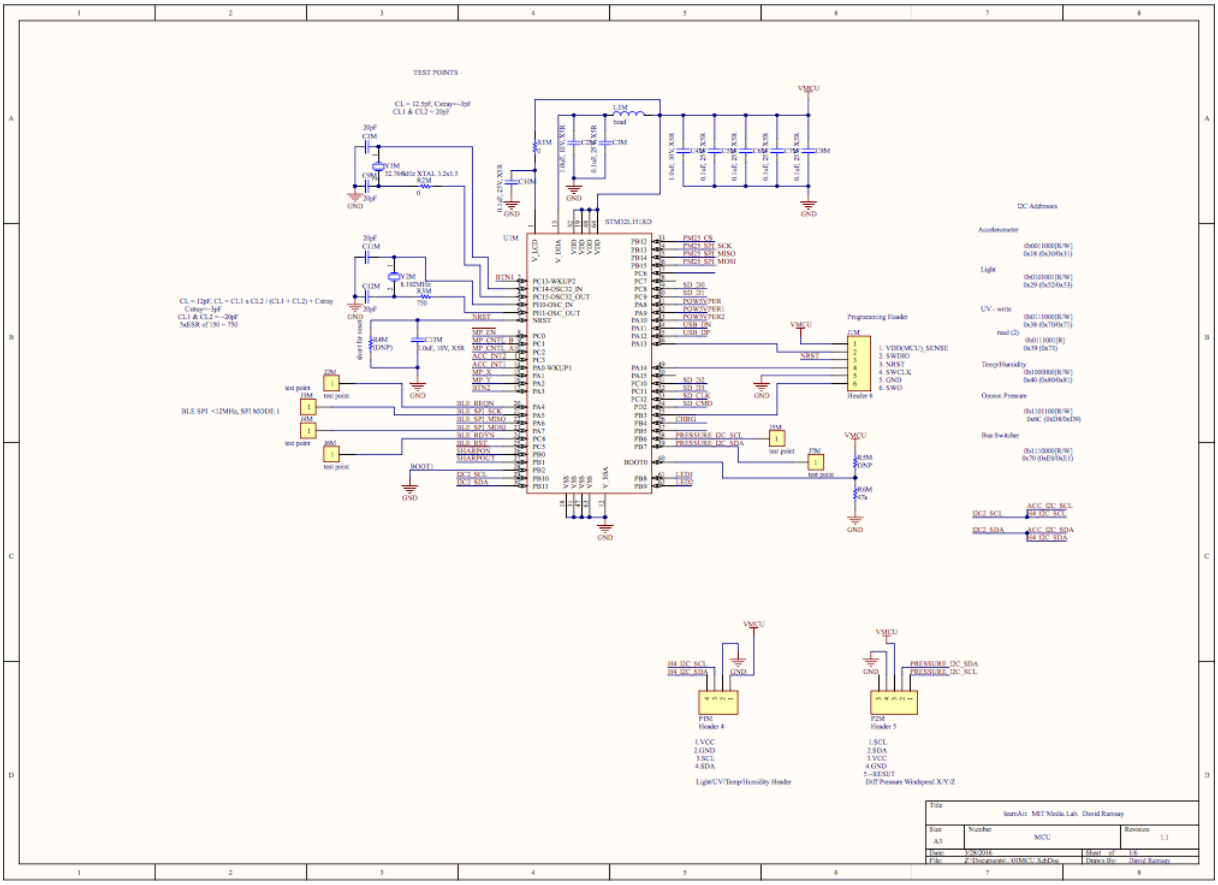

temp/humd, diff pressure, alphasense gas, pm2.5, sensirion tube
header for alphasense gas
PM2.5 - 5V and SPI
sharp PM connector 3.3V, analog out
MAX4618
may need to put low noise op amp with gain of 5/3.3 to match output of MP_X/MP_Y to ADC input
learnAir. MIT Media Lab. David Ramsay
Environmental Sensors
Revision 1.1
Date: 7/16/2016
Sheet of 3/6
File: Z:\Documents\..\03ENV_SENSORS.SchDoc
Drawn By: David Ramsay

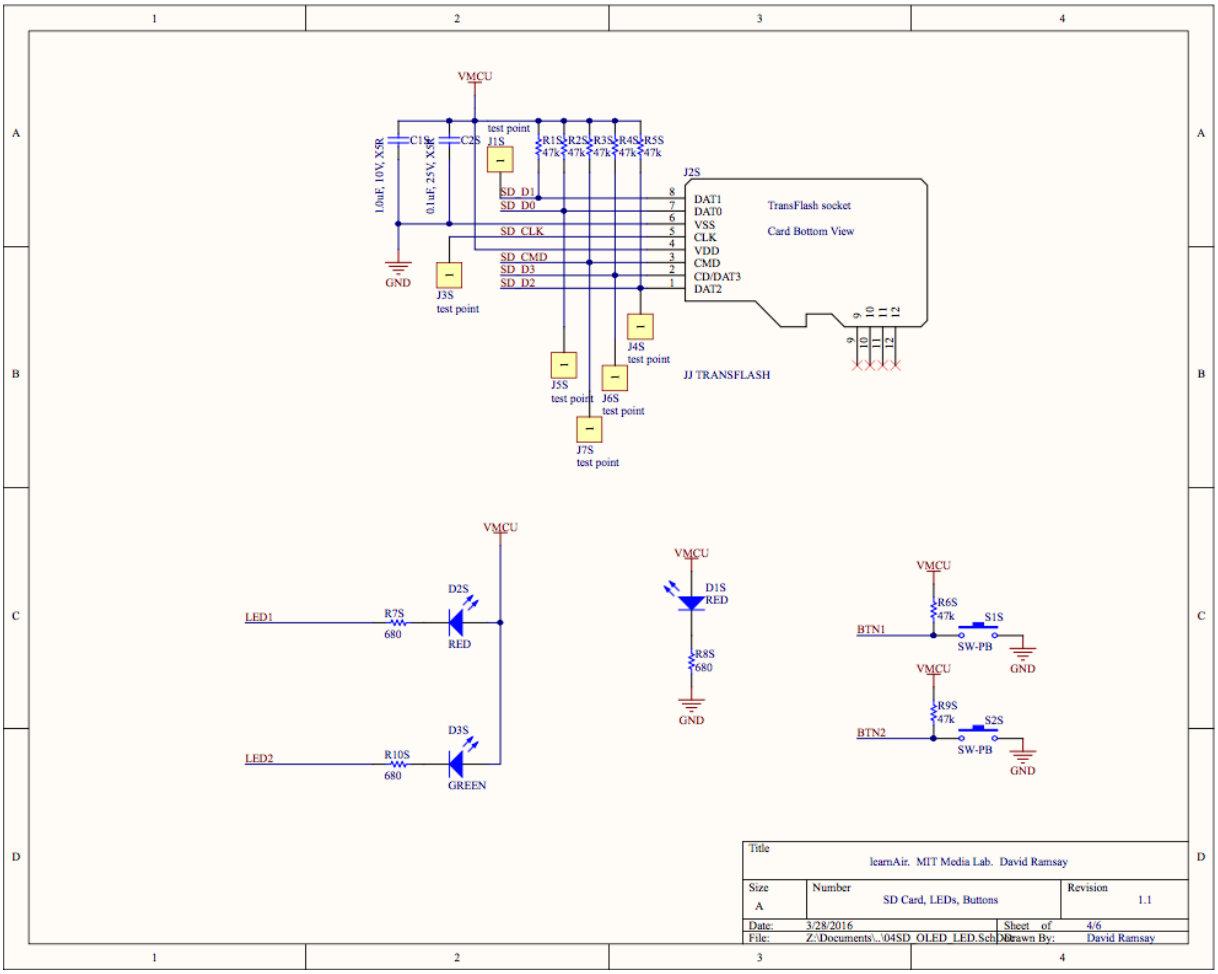

VMCU
C1S
1.0uF, 10V, X5R
C2S
0.1uF, 25V, X5R
GND
J3S
test point
J2S
DAT1
DAT0
VSS
CLK
VDD
CMD
CD/DAT3
DAT2
TransFlash socket
Card Bottom View
SD_DO
SD_SCLK
SD_DI
SD_CS
J5S
test point
J6S
test point
J7S
test point
JJ TRANSFLASH
LED1
R7S
680
D2S
RED
LED2
R10S
680
D3S
GREEN
D1S
RED
R8S
680
BTN1
R6S
47k
S1S
SW-PB
BTN2
R9S
47k
S2S
SW-PB
Title
learnAir. MIT Media Lab. David Ramsay
Size
A
Number
SD Card, LEDs, Buttons
Revision
1.1
Date:
7/16/2016
Sheet of
4/6
File:
Z:\Documents\...\04SD_OLED_LED.SchDoc
Drawn By:
David Ramsay


C1B
12pF, 50V, NP0, 5%
C2B
12pF, 50V, NP0, 5%
Y1B
16.000MHz
GND
L1B
15nH, Q>35, 5%
C4B
1.0uF, 10V, X5R
L2B
10uH, 300mA
R1B
22k, 1%
C3B
1nF, 50V, X7R
VMCU
C5B
0.1uF, 25V, X5R
C6B
0.1uF, 25V, X5R
C7B
0.033uF, 16V, X7R
J1B
test point
nRF8001
Cannot open file
P:\protel_lib\S
CH_Nordic_lo
U2B
nRF8001
BLE_VDDPA
BLE_RST
A1B
ANT
BLE_ANT
U1B
2450BM14A0002
C8B
2200pF, 50V, X7R
J2B
test point
C9B
0.1uF, 25V, X5R
BLE_SPI_SCK
BLE_RFON
BLE_SPI_MOSI
BLE_SPI_MISO
BLE_RDYN
J3B
test point
J4B
test point
Title
learnAir. MIT Media Lab. David Ramsay
Size
A
Number
BLE
Revision
1.1
Date:
7/16/2016
Sheet of
5/6
File:
Z:\Documents\...\05BLE.SchDoc
Drawn By:
David Ramsay


RES TO Ground, SCL/SDA pulled up
I2C address: b001100[SA0][W/R]
pull CS high for I2C
VMCU
1.0uF, 10V, X5R
C1A
0.1uF, 25V, X5R
C2A
GND
J1A test point
R1A DNP
R2A DNP
ACC_I2C_SCL
U1A LIS2DH12
SCL
CS
SA0
SDA
VDD
VDD_IO
RES
GND
INT2
INT1
11 ACC_INT2
12 ACC_INT1
R3A DNP
R4A DNP
R5A DNP
R7A DNP
R6A DNP
ACC_I2C_SDA
test point
J2A
Title
learnAir. MIT Media Lab. David Ramsay
Size A
Number Accelerometer
Revision 1.1
Date: 7/16/2016
Sheet of 6/6
File: Z:\Documents\..\06ACCELEROMETER.SchDoc
Drawn By: David Ramsay

LearnAir V3 uses the same schematics for peripherals, power, accelerometer, and BLE as for LearnAir V2. The two differing schematics (MCU and SD Card) are shown below.





Firmware

Below is the firmware for LearnAir V1. Code written for LearnAir V2 and V3 is under development on github at `http://github.com/dramsay9/learnair`.

```
#include "RTClib.h"
#include "SD.h"
#include "SPI.h"
#include <Wire.h>

//RTC
RTC_DS1307 RTC;
DateTime startTime;

//SD Card
```

```
File sensorData;

//Sharp Dust Sensor
int dustPin=0;
int dustVal=0;
int ledPower=11;
int delayTime=280;
int delayTime2=40;

//Pressure Sensor
unsigned int pressureVal = 0;

//Alphasense Sensor
int asensePin=1;
int asenseValS1W=0;
int asenseValS1A=0;
int asenseValS2W=0;
int asenseValS2A=0;
int asenseValS3W=0;
int asenseValS3A=0;
int asenseValTemp=0;
int aSelAPin=10;
int aSelBPin=9;
int aSelCPin=8;



void setup() {

  //RTC
  RTC.begin();

  if (! RTC.isrunning()) {
    RTC.adjust(DateTime(__DATE__, __TIME__));
  }

  //pressure I2C bus
  Wire.begin();
  Wire.beginTransmission(0x6C);
  Wire.write(byte(0x0B));
  Wire.write(byte(0x00));
  Wire.endTransmission();

  Serial.begin(9600);
```

```
55
56
57
58    //SD Card
59    Serial.print("Initializing␣SD␣card...");
60    // see if the card is present and can be initialized:
61    if (!SD.begin(10, 11, 12, 13)) {
62      Serial.println("Card␣failed,␣or␣not␣present");
63      return;
64    }
65
66    //Create SD Card Title
67    sensorData = SD.open("data.csv", FILE_WRITE);
68    if (sensorData){
69     sensorData.println("timestamp,␣alphaS1_work,␣alphaS1_aux,␣alphaS2_work,
          ↪ ␣alphaS2_aux,␣alphaS3_work,␣alphaS3_aux,␣alphaTemp,␣sharpDust,␣
          ↪ pressureWind");
70    }
71    sensorData.close();
72
73    Serial.println("card␣initialized.");
74
75    //Sharp Dust Sensor
76    pinMode(ledPower,OUTPUT);
77    pinMode(4, OUTPUT);
78
79    //Alphasense Sensor
80    pinMode(aSelAPin, OUTPUT);
81    pinMode(aSelBPin, OUTPUT);
82    pinMode(aSelCPin, OUTPUT);
83
84    delay(100);
85    Serial.println("SETUP␣DONE.␣SENSOR␣READY.");
86
87  }
88
89
90
91  void loop() {
92
93    //RTC Start
94    startTime = RTC.now();
95
96    sharpSense();
```

```
97    pressureSense();
98    alphaSense();
99
100   //SD Write
101   saveData();
102
103   //Print Written Data
104   printData();
105
106   //Wait 30 seconds
107   delay(30000);
108
109 }
110
111
112 void saveData(){
113
114   sensorData = SD.open("data.csv", FILE_WRITE);
115   if (sensorData){
116
117     sensorData.print(startTime.year(), DEC);
118     sensorData.print('/');
119     sensorData.print(startTime.month(), DEC);
120     sensorData.print('/');
121     sensorData.print(startTime.day(), DEC);
122     sensorData.print('␣');
123     sensorData.print(startTime.hour(), DEC);
124     sensorData.print(':');
125     sensorData.print(startTime.minute(), DEC);
126     sensorData.print(':');
127     sensorData.print(startTime.second(), DEC);
128     sensorData.print(",");
129
130     sensorData.print(asenseValS1W);
131     sensorData.print(",");
132     sensorData.print(asenseValS1A);
133     sensorData.print(",");
134     sensorData.print(asenseValS2W);
135     sensorData.print(",");
136     sensorData.print(asenseValS2A);
137     sensorData.print(",");
138     sensorData.print(asenseValS3W);
139     sensorData.print(",");
140     sensorData.print(asenseValS3A);
```

```
141     sensorData.print(",");
142     sensorData.print(asenseValTemp);
143     sensorData.print(",");
144 
145     sensorData.print(dustVal);
146     sensorData.print(",");
147     sensorData.println(pressureVal);
148 
149     sensorData.close(); // close the file
150 
151     Serial.println("Wrote␣to␣file.");
152   }
153   else{
154     Serial.println("Error␣writing␣to␣file!");
155   }
156 
157 }
158 
159 
160 void sharpSense(){
161 
162   digitalWrite(ledPower,LOW); // power on the LED
163   delayMicroseconds(delayTime);
164   dustVal=analogRead(dustPin); // read the dust value
165   delayMicroseconds(delayTime2);
166   digitalWrite(ledPower,HIGH); // turn the LED off
167 
168 }
169 
170 
171 void pressureSense(){
172 
173   Wire.beginTransmission(0x6C);
174   Wire.write(byte(0x00));
175   Wire.write(byte(0xD0));
176   Wire.write(byte(0x40));
177   Wire.write(byte(0x18));
178   Wire.write(byte(0x06));
179   Wire.endTransmission();
180 
181   Wire.beginTransmission(0x6C);
182   Wire.write(byte(0x00));
183   Wire.write(byte(0xD0));
184   Wire.write(byte(0x51));
```

```
185   Wire.write(byte(0x2C));
186   Wire.endTransmission();
187
188   Wire.beginTransmission(0x6C);
189   Wire.write(byte(0x07));
190   Wire.endTransmission();
191
192   Wire.requestFrom(0x6C, 2);    // request 2 bytes from Pressure Sensor
193
194   if (2 <= Wire.available())  {  // if two bytes were received
195     pressureVal = Wire.read();  // receive high byte (overwrites previous
          ↪ reading)
196     pressureVal = pressureVal << 8;    // shift high byte to be high 8
          ↪ bits
197     pressureVal |= Wire.read(); // receive low byte as lower 8 bits
198   }
199
200 }
201
202
203 void alphaSense() {
204
205   const int delayNum = 10;
206
207   digitalWrite(aSelCPin,LOW);
208   digitalWrite(aSelBPin,LOW);
209   digitalWrite(aSelAPin,HIGH);
210   delayMicroseconds(delayNum);
211   asenseValS1W=analogRead(asensePin);
212
213   digitalWrite(aSelCPin,LOW);
214   digitalWrite(aSelBPin,HIGH);
215   digitalWrite(aSelAPin,LOW);
216   delayMicroseconds(delayNum);
217   asenseValS1A=analogRead(asensePin);
218
219   digitalWrite(aSelCPin,LOW);
220   digitalWrite(aSelBPin,HIGH);
221   digitalWrite(aSelAPin,HIGH);
222   delayMicroseconds(delayNum);
223   asenseValS2W=analogRead(asensePin);
224
225   digitalWrite(aSelCPin,HIGH);
226   digitalWrite(aSelBPin,LOW);
```

```
227   digitalWrite(aSelAPin,LOW);
228   delayMicroseconds(delayNum);
229   asenseValS2A=analogRead(asensePin);
230 
231   digitalWrite(aSelCPin,HIGH);
232   digitalWrite(aSelBPin,LOW);
233   digitalWrite(aSelAPin,HIGH);
234   delayMicroseconds(100);
235   asenseValS3W=analogRead(asensePin);
236 
237   digitalWrite(aSelCPin,HIGH);
238   digitalWrite(aSelBPin,HIGH);
239   digitalWrite(aSelAPin,LOW);
240   delayMicroseconds(delayNum);
241   asenseValS3A=analogRead(asensePin);
242 
243   digitalWrite(aSelCPin,HIGH);
244   digitalWrite(aSelBPin,HIGH);
245   digitalWrite(aSelAPin,HIGH);
246   delayMicroseconds(delayNum);
247   asenseValTemp=analogRead(asensePin);
248 
249 }
250 
251 
252 void printData(){
253 
254 
255     Serial.print(startTime.year(), DEC);
256     Serial.print('/');
257     Serial.print(startTime.month(), DEC);
258     Serial.print('/');
259     Serial.print(startTime.day(), DEC);
260     Serial.print(' ');
261     Serial.print(startTime.hour(), DEC);
262     Serial.print(':');
263     Serial.print(startTime.minute(), DEC);
264     Serial.print(':');
265     Serial.print(startTime.second(), DEC);
266     Serial.println("-------------------------------------");
267 
268 
269     Serial.print("Alphasense Sensor 1.  Working: ");
270     Serial.print(asenseValS1W);
```

```
271     Serial.print("␣␣Aux:␣␣");
272     Serial.println(asenseValS1A);
273     Serial.print("Alphasense␣Sensor␣2.␣␣Working:␣");
274     Serial.print(asenseValS2W);
275     Serial.print("␣␣Aux:␣␣");
276     Serial.println(asenseValS2A);
277     Serial.print("Alphasense␣Sensor␣3.␣␣Working:␣");
278     Serial.print(asenseValS3W);
279     Serial.print("␣␣Aux:␣␣");
280     Serial.println(asenseValS3A);
281
282     Serial.print("Alphasense␣Temp:␣");
283     Serial.println(asenseValTemp);
284
285     Serial.print("Sharp␣Dust:␣");
286     Serial.println(dustVal);
287     Serial.print("Pressure:␣");
288     Serial.println(pressureVal);
289
290 }
```

Hardware Analysis

Figure 57 shows the windspeed measurement comparison between our conditioned pressure sensor measurement and the MassDEP windspeed measurement. Within $\pm 5\%$ is denoted with green highlights.

Figure 58 shows the wind direction angle over the course of a day, with an indication of how closely the conditioned wind pressure sensor reading matched the actual windspeed. Green highlighting indicates that our windspeed measurement was within $\pm 5\%$ of the MassDEP reading. This plot is useful to look for systemic errors– are there any wind directions where we consistently are accurate in our measurment? Are there any wind directions where we're inaccurate? It appears there are no obvious relationships between wind direction and accuracy from this graph. However, there are interesting relationships between wind direction and error– please see Chapter 5.

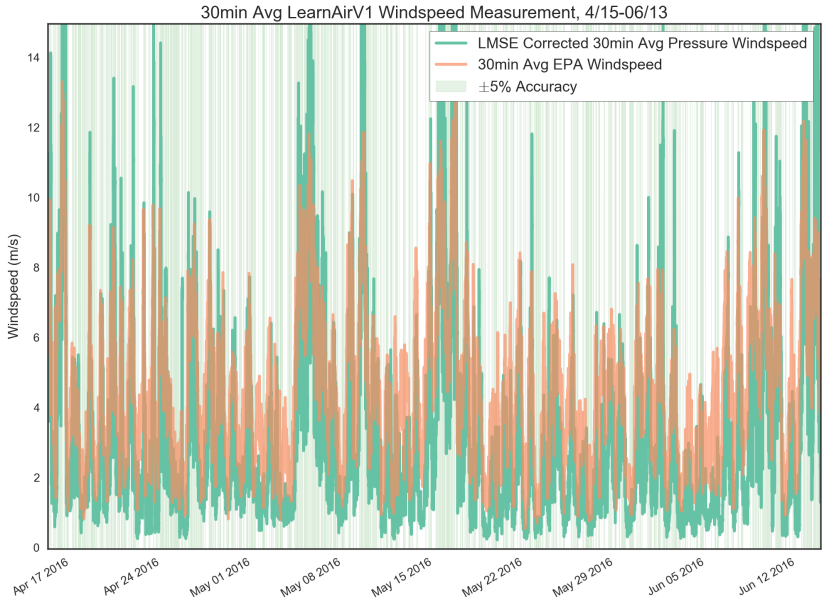


Figure 57: Wind Speed Measurement with 10% Accuracy

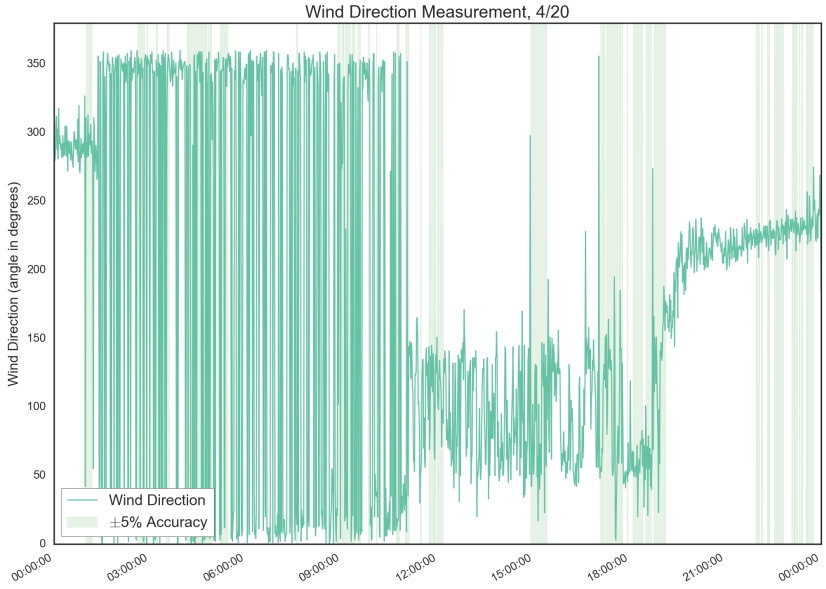


Figure 58: Wind Direction with 10% Accuracy WindSpeed Measurements Denoted

Appendix C - ChainAPI Code

The original instance of ChainAPI can be found and installed from `https://github.com/ResEnv/chain-api`, and a browsable version of the Tidmarsh data backend in ChainAPI can be found at `http://chain-api.media.mit.edu/`. It is well worth checking out some of the interesting visualizations at `http://tidmarsh.media.mit.edu/viz/` and some of the interesting projects built on top of ChainAPI at `http://tidmarsh.media.mit.edu/`.

The air quality version of ChainAPI is an on-going work, and the latest version can be found on github at `http://github.com/dramsay9/chain-api`. A testing and development version of this instance is generally available for browsing at `http://learnair.media.mit.edu:8000`. The pre-processing and machine learning scripts can be found in a jupyter notebook at `http://github.com/dramsay9/learnair-data-crunching`, while the main chain tools are found in their corresponding git repository: `http://github.com/dramsay9/chaincrawler`, `http://github.com/dramsay9/chainlearnairdata`, and `http://github.com/dramsay9/chaindataprocessor`.

Appendix D - Machine Learning

Test Conditions and Data Summary

The following charts show trends in precipitation, ambient pressure, cloud cover, dew, and light level over the course of our two month MassDEP co-location test.

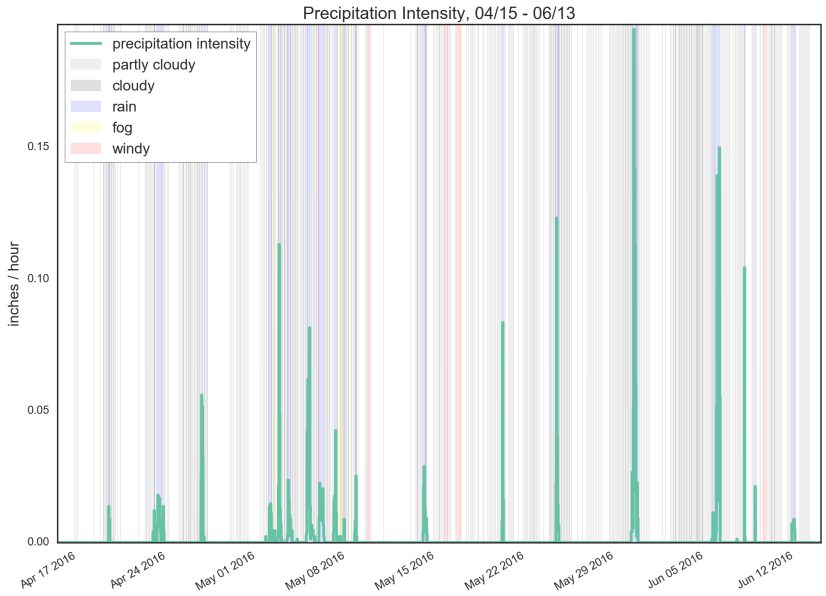


Figure 59: Precipitation Intensity during Test Period

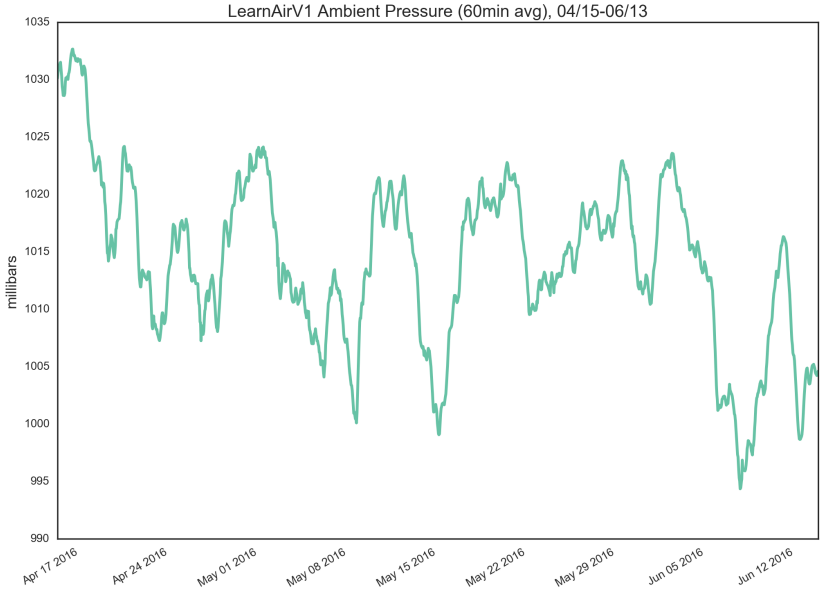


Figure 60: Ambient Pressure during Test Period

Table 27: Most Frequent Weather During Co-location Test

Weather Type	# Hours During Test
clear hours	901
partly cloudy hours	292
cloudy hours	103
raining hours	96
windy hours	25
foggy hours	11

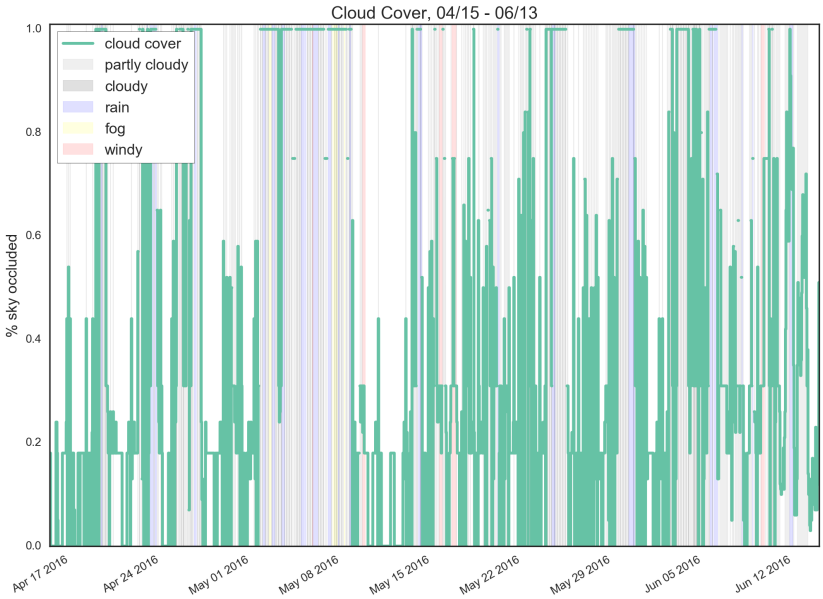


Figure 61: Cloud Cover during Test Period

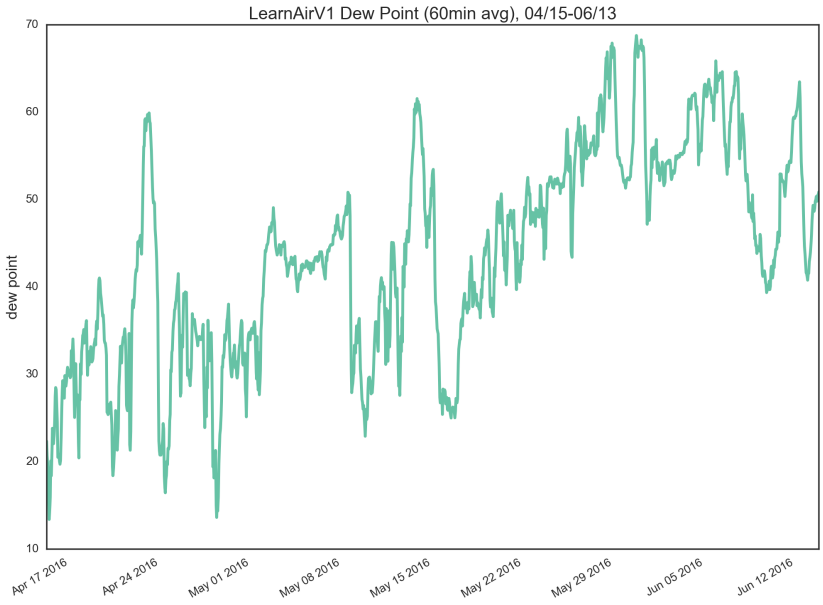


Figure 62: Dew during Test Period

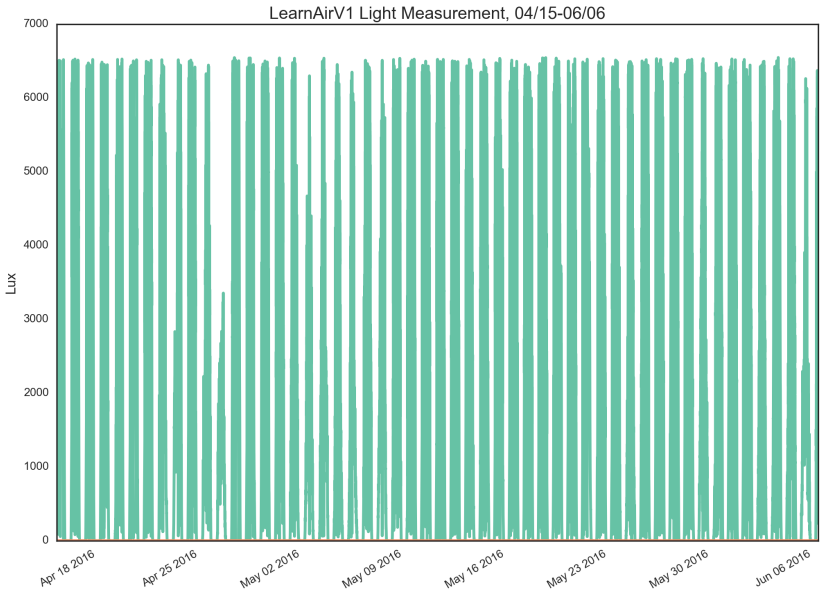


Figure 63: Lux during Test Period

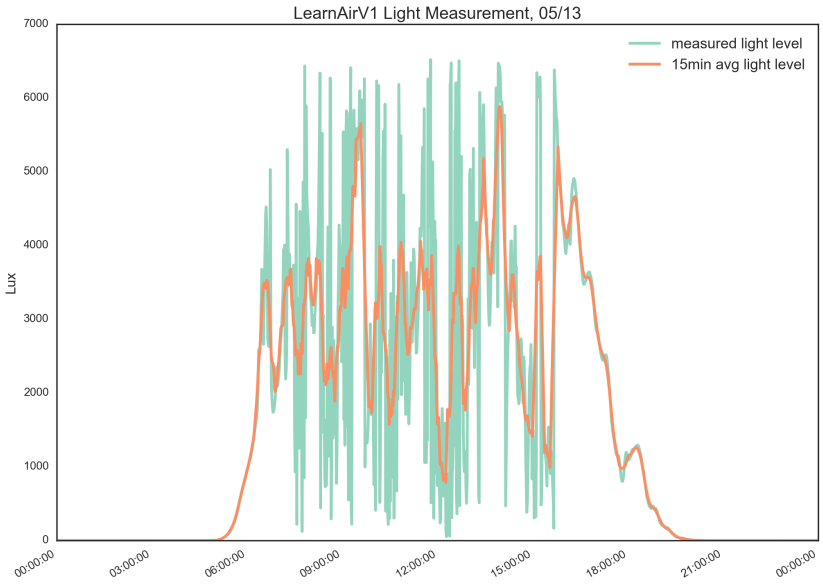


Figure 64: Lux during Test Period

Features

The figure below includes most of the machine learning feature distributions plotted using Weka.

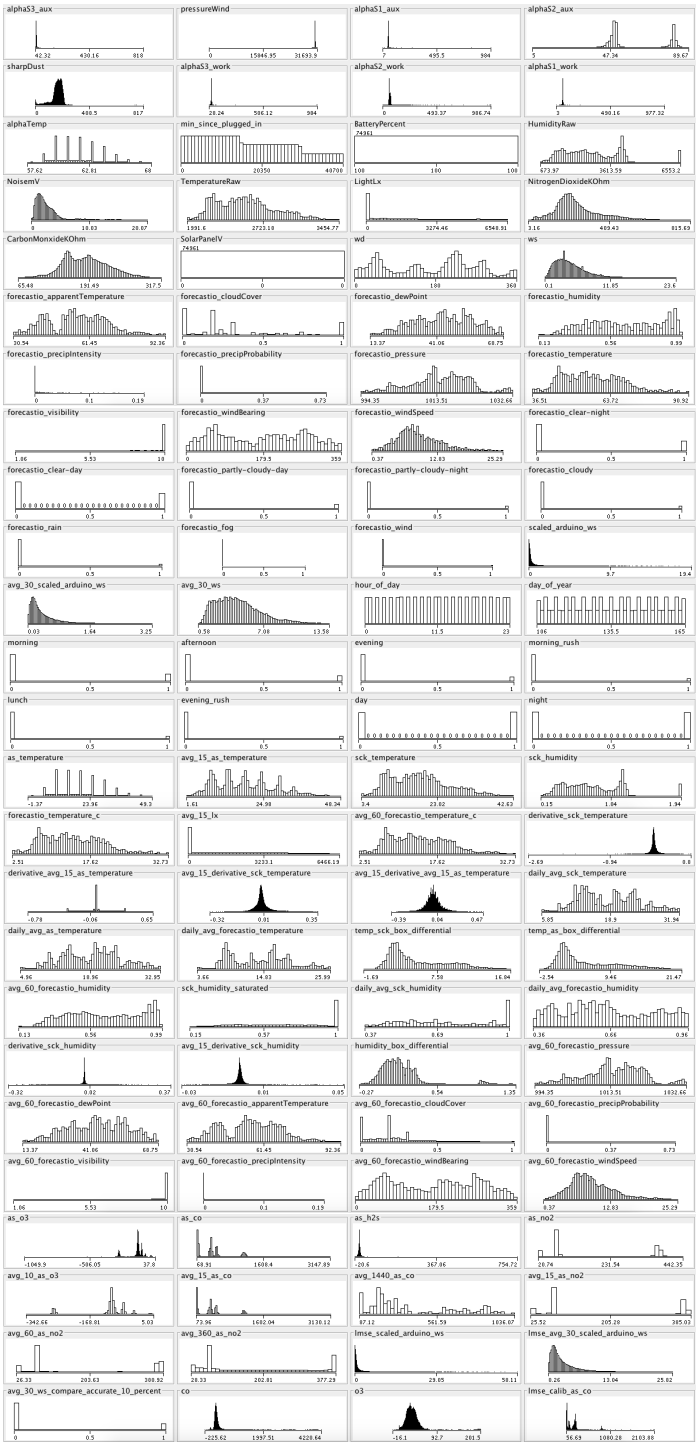


Figure 65: ML feature histograms plotted with WEKA Tool

SmartCitizen CO

Following are additional plots from the SmartCitizen CO test outlining in more detail the LMSE calibration, the accuracy with a tighter 5% threshold, a visualization of the prediction accuracy and confidence, and the top 15 random forest selected features.

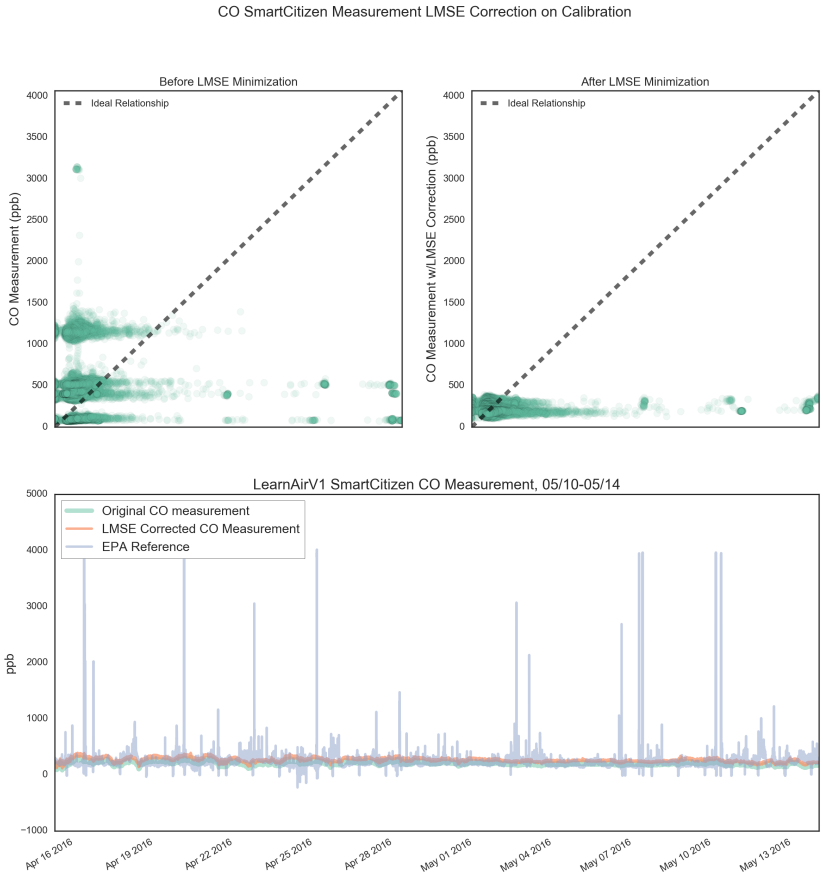


Figure 66: SmartCitizen CO after LMSE Calibration

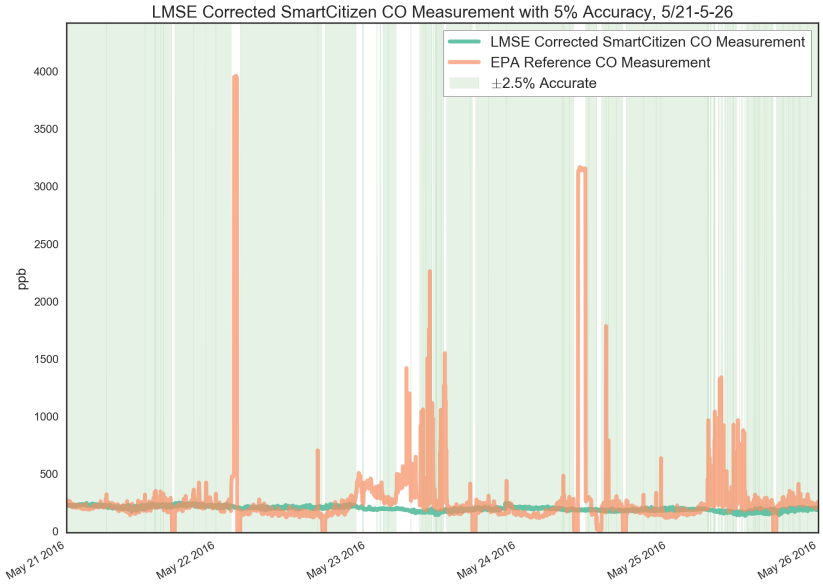


Figure 67: SmartCitizen CO with 5% Accuracy Threshold

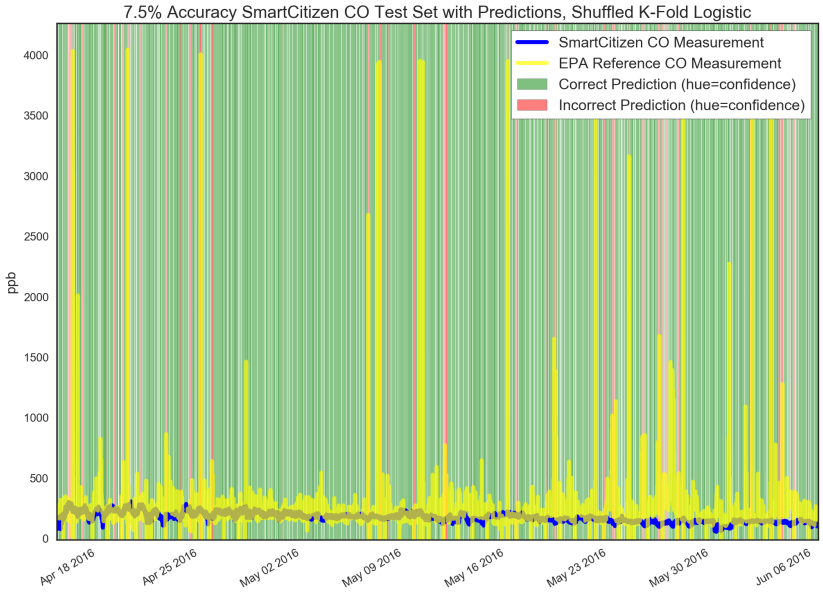


Figure 68: SmartCitizen CO Prediction Accuracy

Table 28: Top 15 Features from Random Forest for SmartCitizen CO, used in Pruned Logistic Regression

Feature	Importance
bkcarbon	0.027481618644
avg_60_bkcarbon	0.0265308524121
avg_720_bkcarbon	0.0231734007362
avg_1440_bkcarbon	0.0213230536622
avg_60_forecastio_windSpeed	0.0155772873357
min_since_plugged_in	0.0151174982516
temp_sck_box_differential	0.0148499597107
avg_60_forecastio_windBearing	0.014573874136
daily_avg_forecastio_humidity	0.0145367615821
avg_60_forecastio_dewPoint	0.0138511147354
avg_60_forecastio_pressure	0.0138476329536
daily_avg_sck_temperature	0.0138353139286
avg_30_ws	0.0136031033823
daily_avg_sck_humidity	0.0135231176757
avg_720_lmse_scaled_sharpDust	0.0132885608127

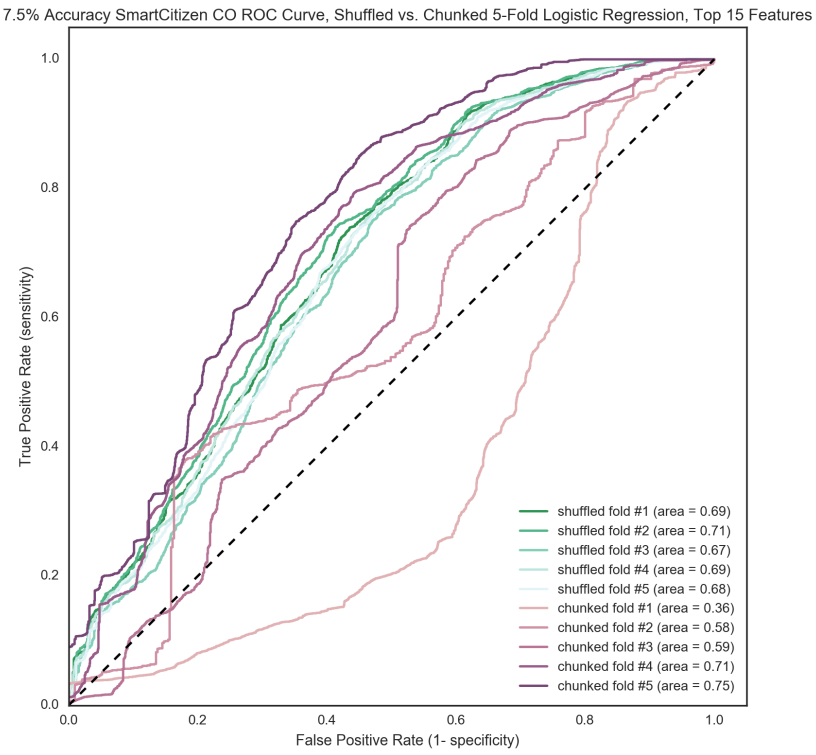


Figure 69: SmartCitizen CO ROC Using Top 15 Features

SmartCitizen NO2

Following are additional plots from the SmartCitizen NO2 test outlining in more detail the LMSE calibration, the accuracy with a tighter 4% threshold, and the top 15 random forest selected features.

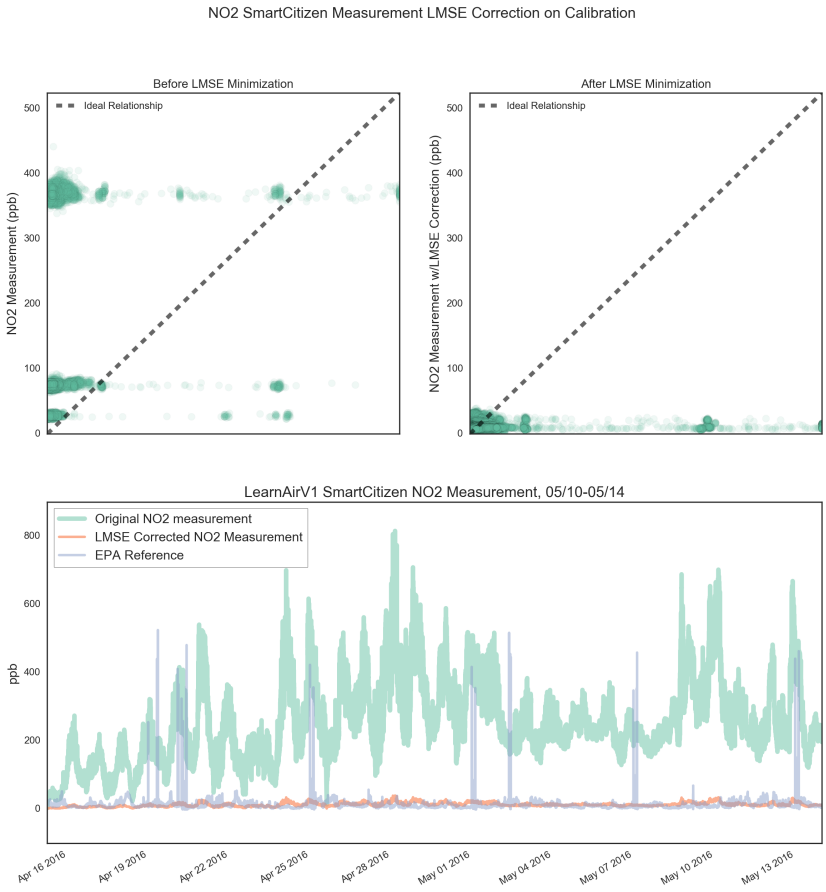


Figure 70: SmartCitizen NO2 after LMSE Calibration

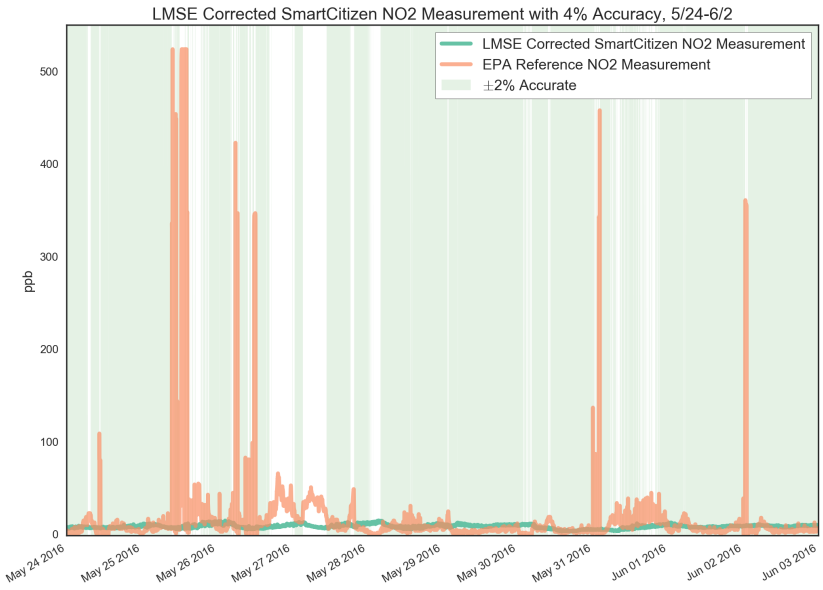


Figure 71: SmartCitizen NO2 with 4% Accuracy Threshold

Table 29: Top 15 Features from Random Forest for SmartCitizen NO2, used in Pruned Logistic Regression

Feature	Importance
bkcarbon	0.0459890536212
avg_60_bkcarbon	0.0433384018273
avg_720_bkcarbon	0.024690468695
avg_1440_bkcarbon	0.0210702674105
avg_60_forecastio_windSpeed	0.0207714428351
min_since_plugged_in	0.0173782542533
avg_60_forecastio_windBearing	0.0172875801677
forecastio_windSpeed	0.0170176630128
avg_1440_lmse_calib_as_co	0.0162266191466
daily_avg_sck_humidity	0.0160827543221
avg_60_forecastio_pressure	0.0157403595739
avg_720_lmse_scaled_sharpDust	0.0154263296837
avg_1440_lmse_scaled_sharpDust	0.0153038668128
daily_avg_as_temperature	0.0151434934355
daily_avg_forecastio_temperature	0.0148922895233

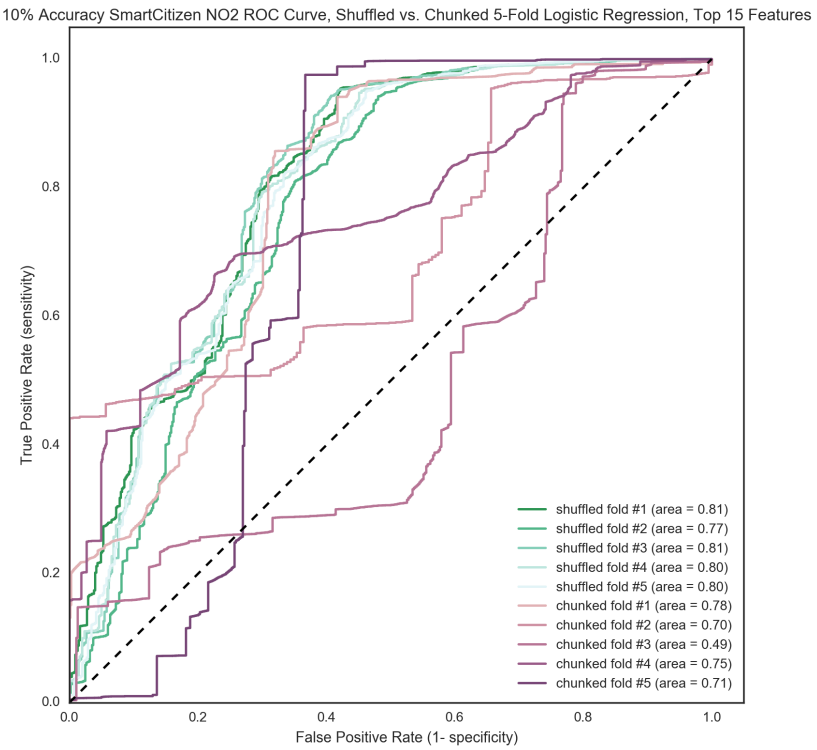


Figure 72: SmartCitizen NO2 ROC Using Top 15 Features

Sharp Dust Sensor

Following are additional plots from the Sharp dust sensor test outlining the complete raw data, a visualization of the prediction accuracy and confidence, and the top 15 random forest selected features.

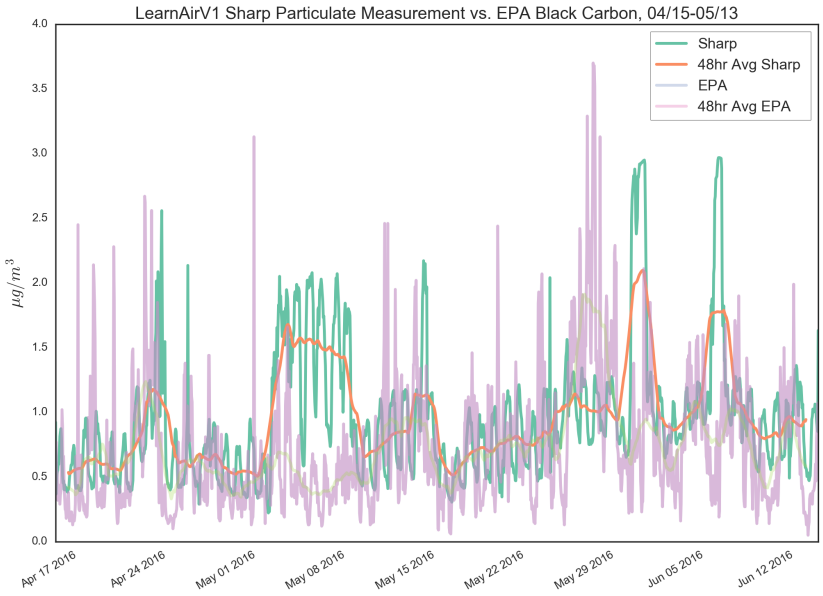


Figure 73: Sharp Raw Particulate Data

Following are ROC curves for the Sharp sensor using just the top 15 features (for both the hour and 48 hour averaged data), as well as ROC curves and features for tighter tolerances (15% instead of 30% of full-scale).

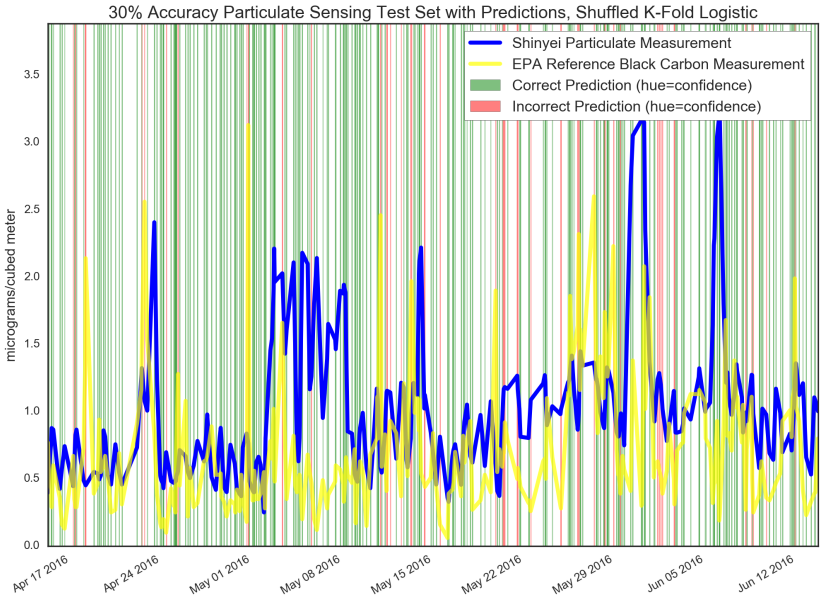


Figure 74: Sharp Particulate Prediction Accuracy

Table 30: Top 15 Features from Random Forest for Sharp Sensor, used in Pruned Logistic Regression

Feature	Importance
scaled_sharpDust	0.039935725943
avg_12_scaled_sharpDust	0.0390147943972
sharpDust	0.0390147211728
lmse_scaled_sharpDust	0.0381632767126
avg_48_scaled_sharpDust	0.0225005941711
lmse_avg_48_scaled_sharpDust	0.0207695248823
sck_humidity	0.0163292725576
Humidity (% RAW)	0.0162400825573
no2	0.0149758603207
daily_avg_sck_humidity	0.0138699992039
daily_avg_forecastio_humidity	0.0132135840929
humidity_box_differential	0.0119641893085
co	0.0118968560369
sck_humidity_saturated	0.0103888721788
avg_60_forecastio_humidity	0.0102980544091

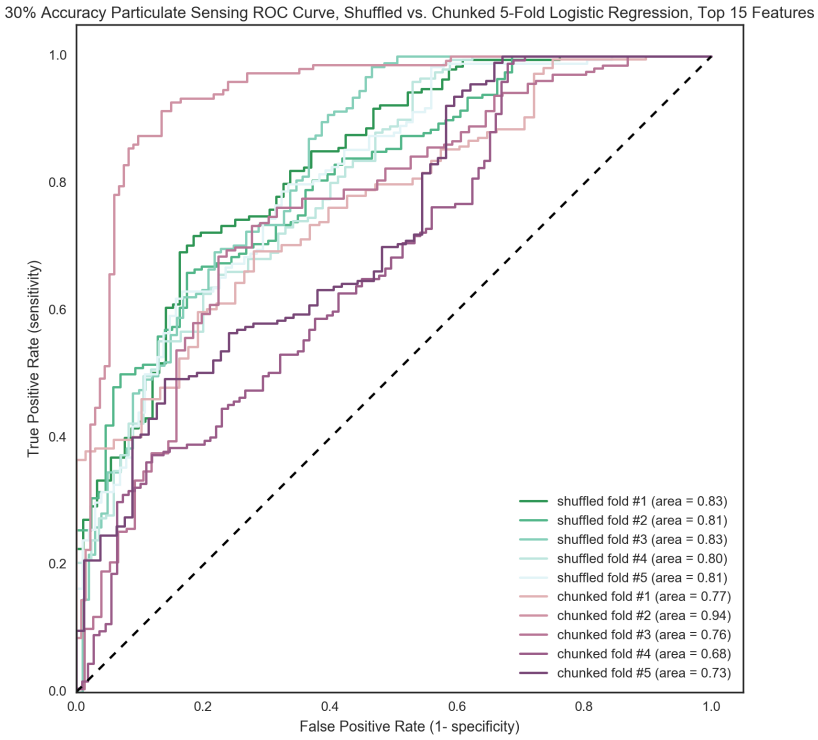


Figure 75: Sharp Particulate ROC Using Top 15 Features

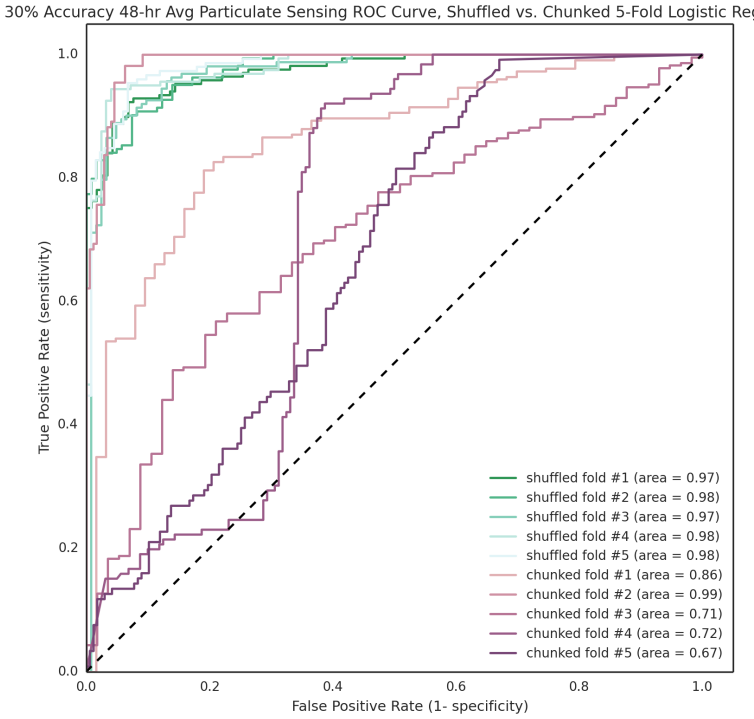


Figure 76: 48-hour Average Sharp Particulate ROC

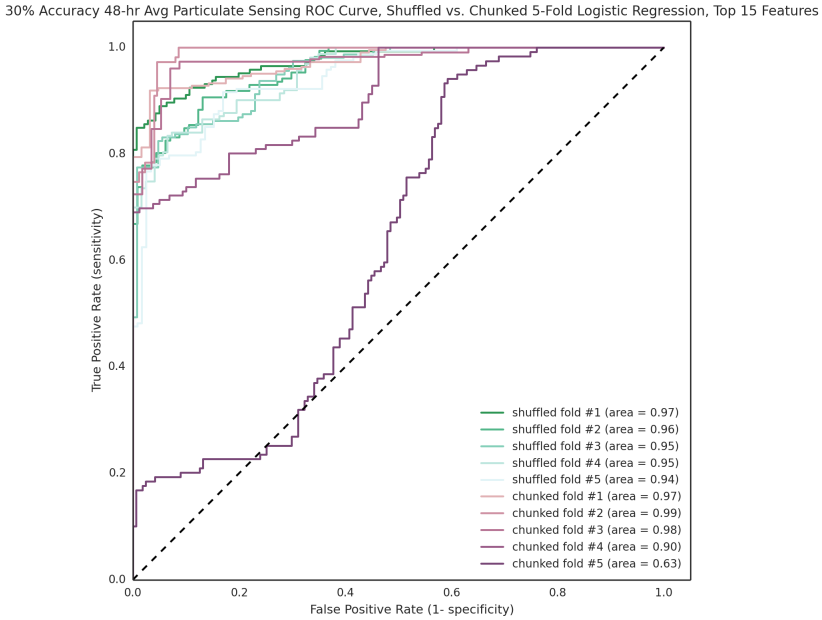


Figure 77: 48-hour Average Sharp Particulate ROC Using Top 15 Features

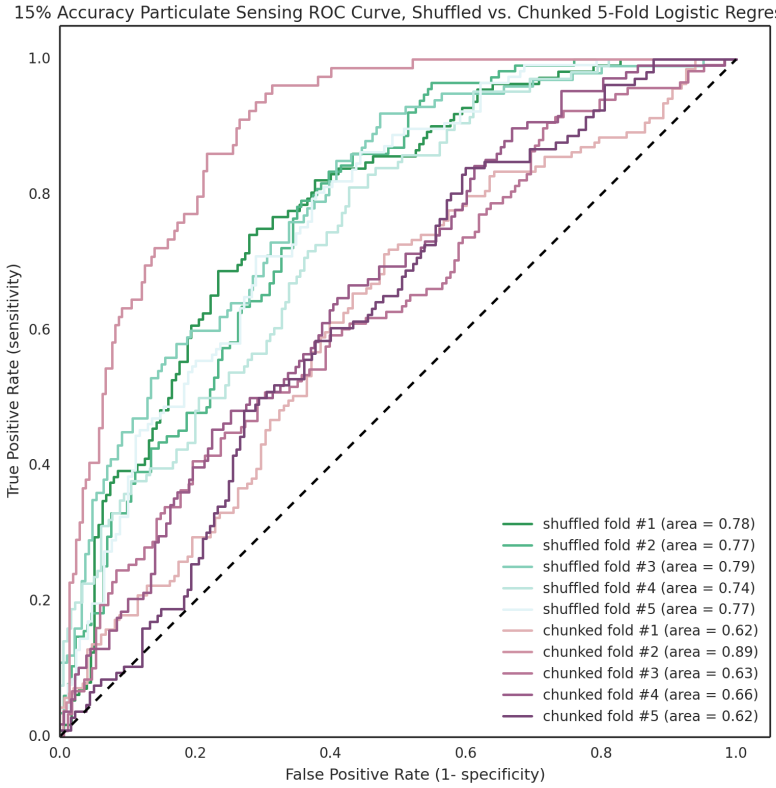


Figure 78: Reduced Tolerance Sharp Particulate ROC

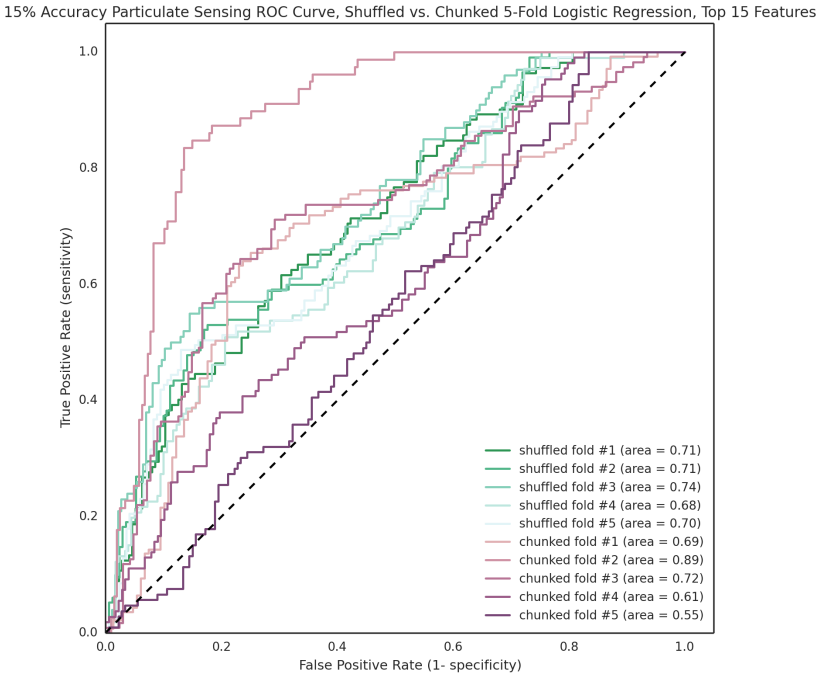


Figure 79: Reduced Tolerance Sharp Particulate ROC Using Top 15 Features

AlphaSense CO

Following are additional plots from the AlphaSense CO test outlining the complete raw data and LMSE process, the accuracy with a tighter 7.5% threshold, a visualization of the prediction accuracy and confidence, and the top 15 random forest selected features for both sensors tested. Additionally, the ROC plots using just the top 15 features is included for both sensors.

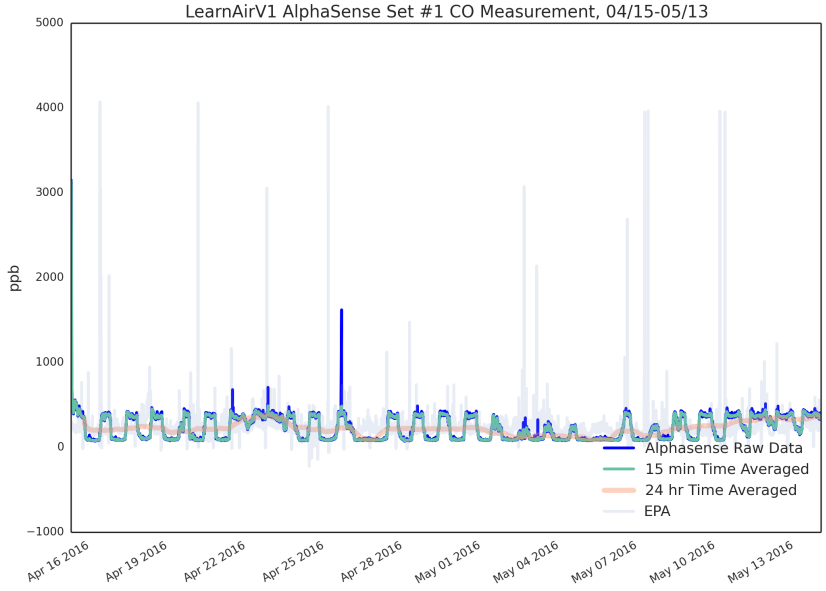


Figure 80: AlphaSense CO Sensor 1 Raw Data

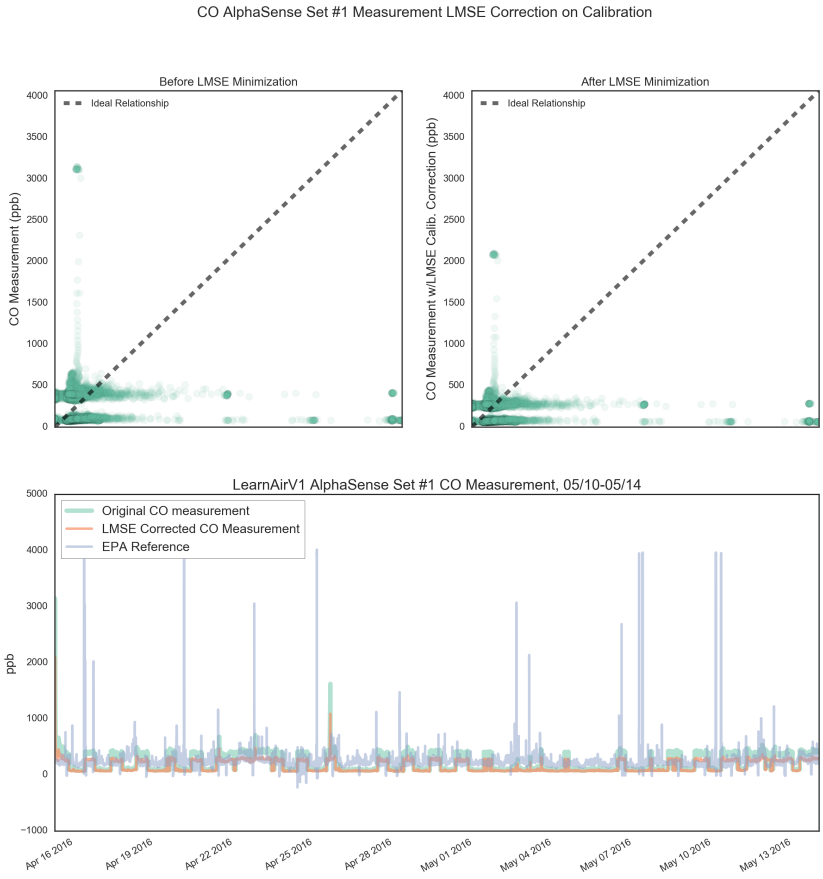


Figure 81: AlphaSense CO Sensor 1 after LMSE Calibration

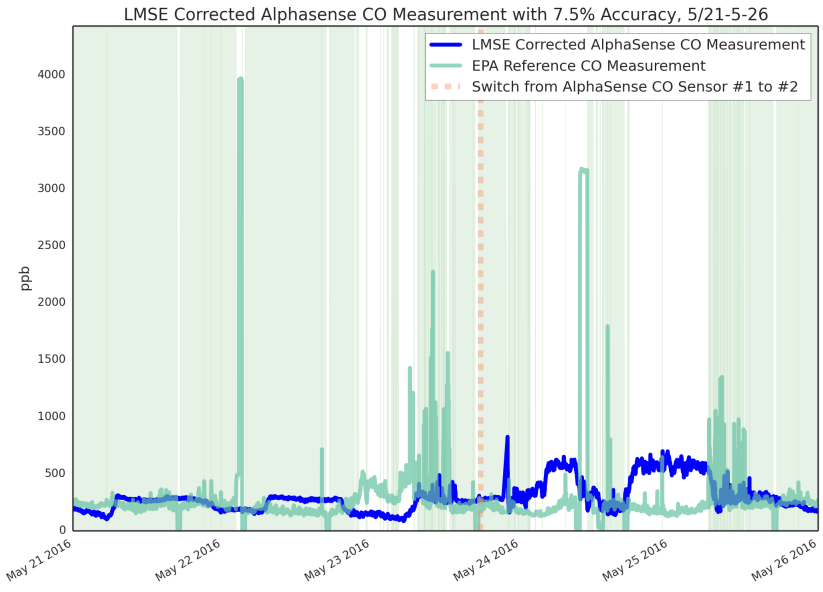


Figure 82: AlphaSense CO Sensor 1 and 2 with 7.5% Accuracy Threshold

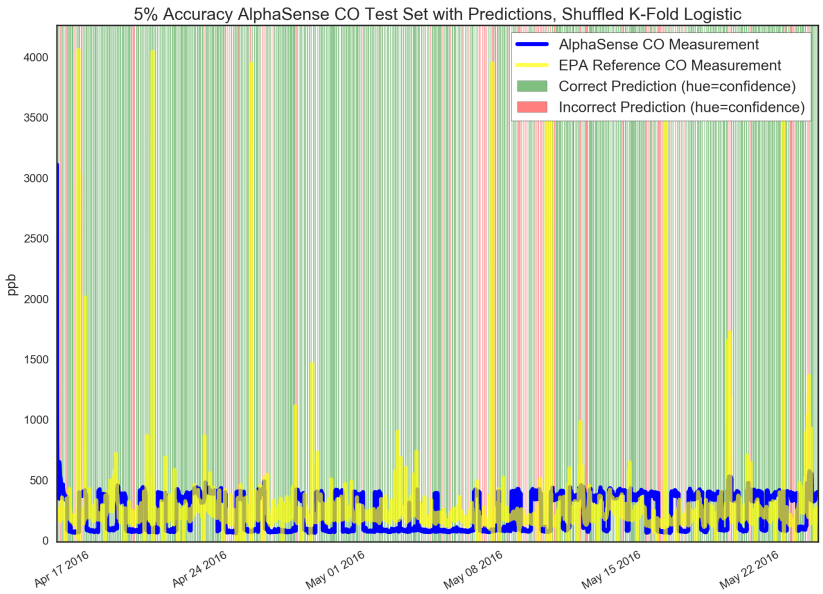


Figure 83: AlphaSense CO Sensor 1 Prediction Accuracy

Table 31: Top 15 Features from Random Forest for CO Sensor 1, used in Pruned Logistic Regression

Feature	Importance
lmse_calib_as_co	0.0331983839664
avg_15_lmse_calib_as_co	0.0331946346979
avg_15_as_co	0.0322602817136
as_co	0.0310204625161
sck_temperature	0.0271023795431
avg_15_as_temperature	0.0251288063362
Temperature (C RAW)	0.024693128613
avg_60_forecastio_temperature$_c$	0.0207050630804
forecastio_temperature_c	0.0192957142054
as_temperature	0.018952457567
avg_60_forecastio_apparentTemperature	0.017952895033
alphaTemp	0.0177934801727
temp_as_box_differential	0.017581796276
forecastio_temperature	0.0159096583245
temp_sck_box_differential	0.0150463135619

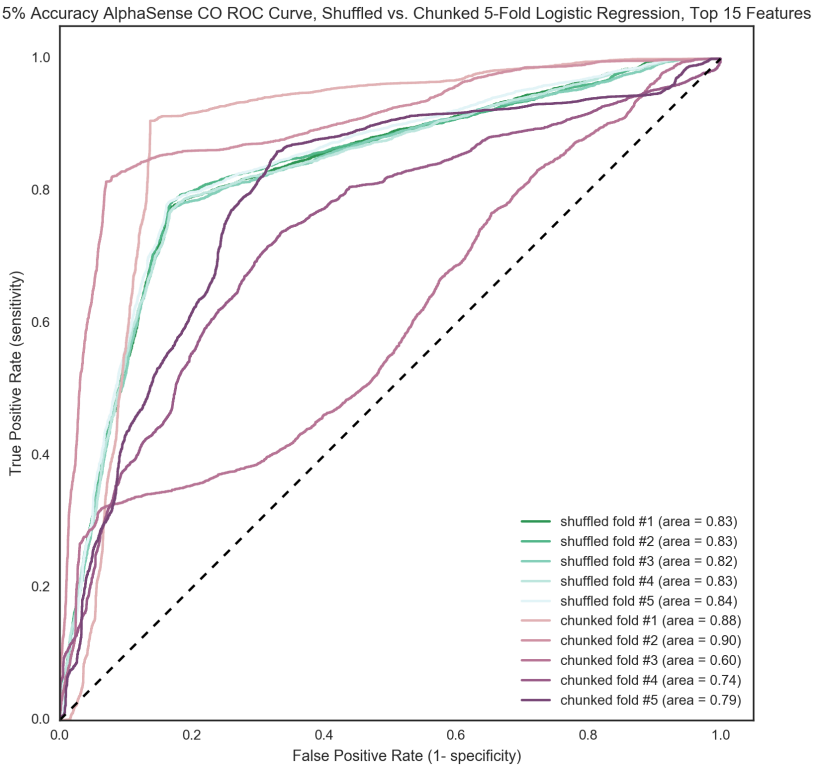


Figure 84: AlphaSense CO Sensor 1 ROC Using Top 15 Features

Table 32: Top 15 Features from Random Forest for CO Sensor 2, used in Pruned Logistic Regression

Feature	Importance
lmse_calib_as_co	0.0518584805682
avg_15_lmse_calib_as_co	0.0404238890793
avg_720_bkcarbon	0.0222537733125
avg_60_bkcarbon	0.0216045744972
avg_1440_bkcarbon	0.0198813295966
as_o3	0.0198510401658
lmse_as_no2	0.0197364055605
avg_10_as_o3	0.0196965727088
bkcarbon	0.0194862747741
as_no2	0.0192353467551
avg_15_lmse_as_no2	0.0180978893662
lmse_avg_15_as_no2	0.0172526534474
avg_15_as_no2	0.0162905415767
avg_15_as_co	0.0158810645781
as_co	0.0158442759727

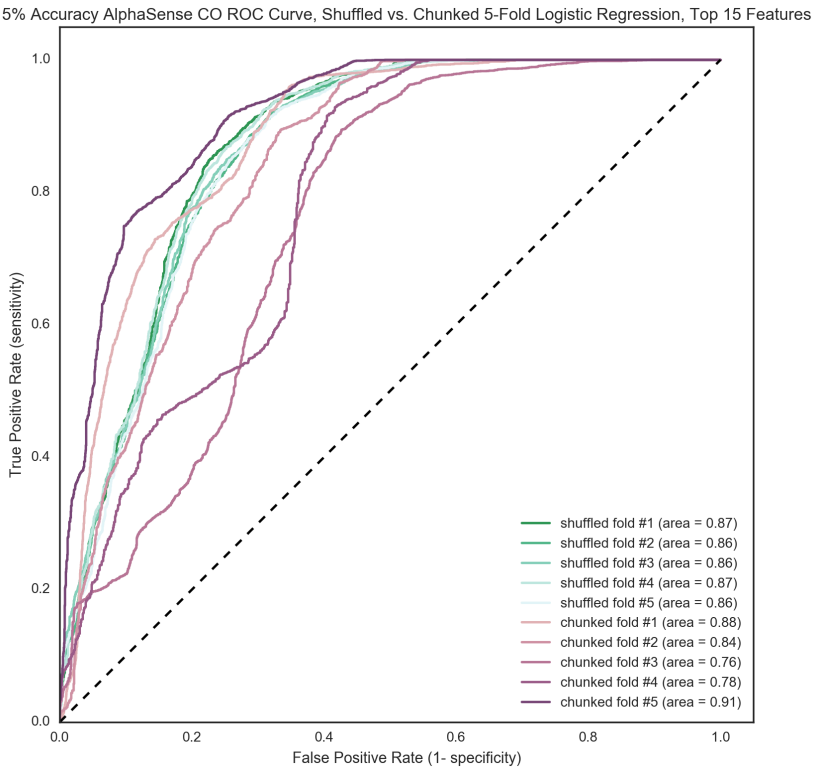


Figure 85: AlphaSense CO Sensor 2 ROC Using Top 15 Features

AlphaSense NO2

Following are additional plots from the AlphaSense NO2 test outlining the complete raw data and LMSE process, the accuracy with a tighter 4% threshold, and the top 15 random forest selected features.

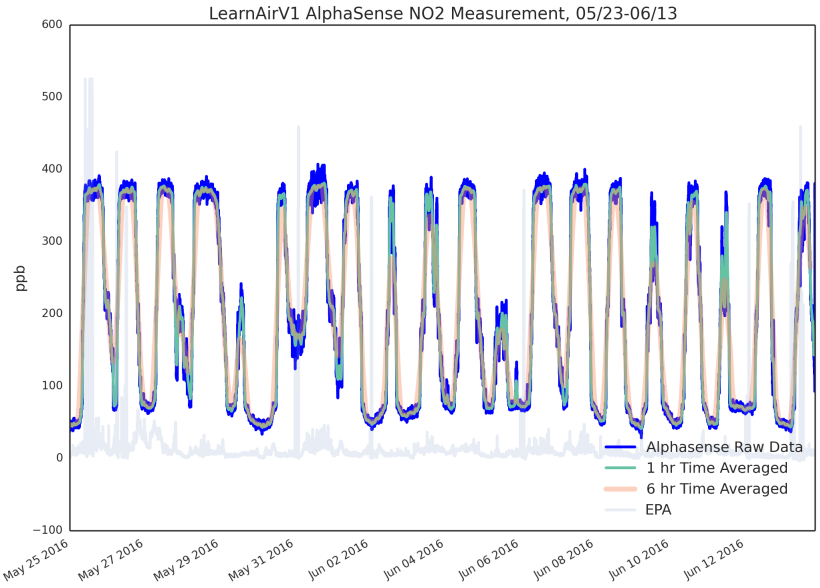


Figure 86: AlphaSense NO2 Raw Data

Figure 89 is the ROC using just the top 15 features to predict NO2.

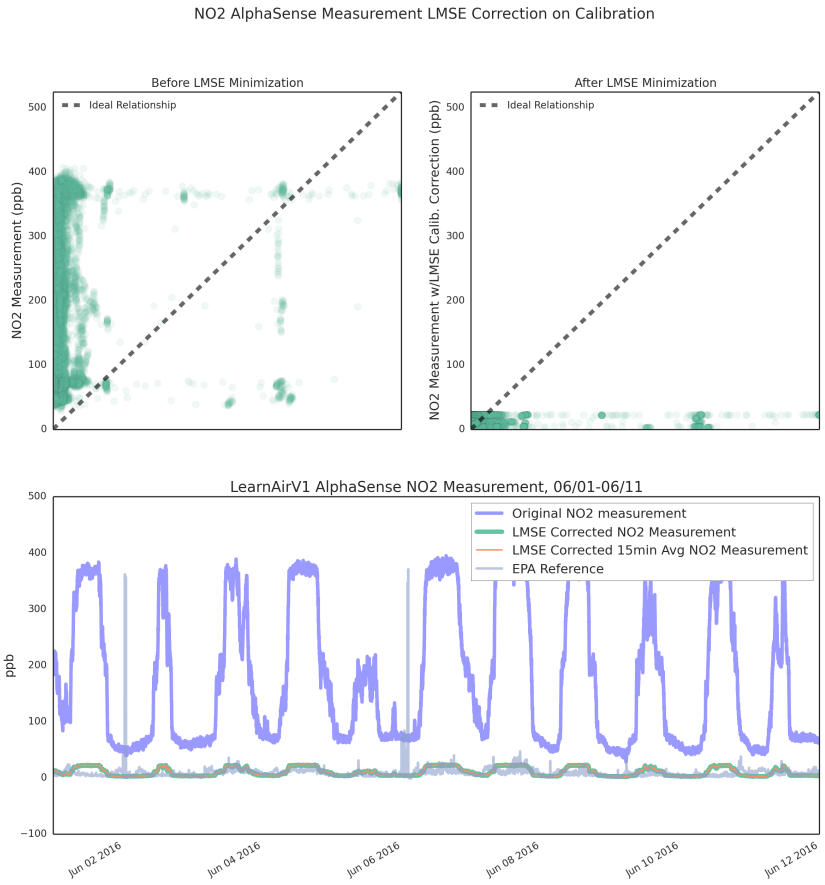


Figure 87: AlphaSense NO_2 after LMSE Calibration

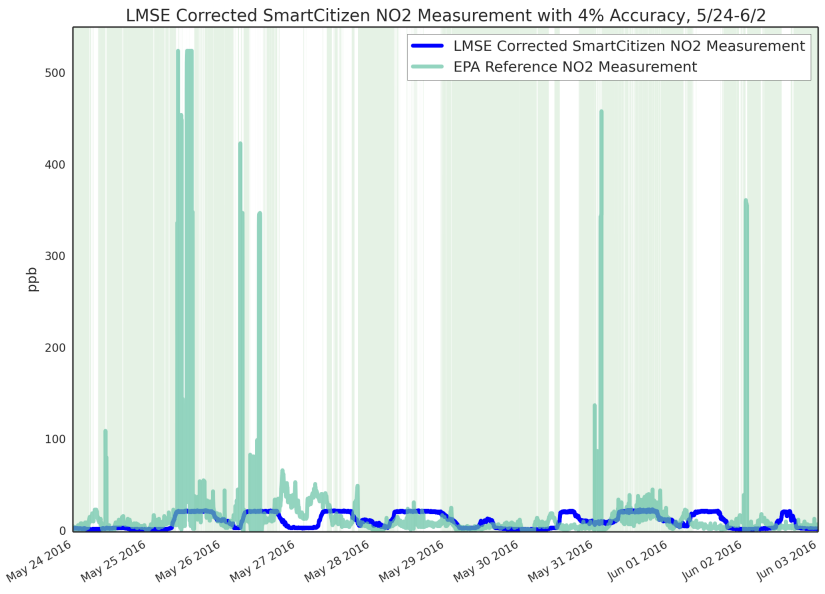


Figure 88: AlphaSense NO_2 with 4% Accuracy Threshold

Table 33: Top 15 Features from Random Forest for AlphaSense NO_2, used in Pruned Logistic Regression

Feature	Importance
avg_60_bkcarbon	0.0422524706607
avg_1440_bkcarbon	0.0417472204692
bkcarbon	0.0385594210158
avg_720_bkcarbon	0.0347584125412
min_since_plugged_in	0.0203302045169
avg_60_forecastio_windSpeed	0.0164269542704
derivative_avg_1440_bkcarbon	0.0162252088513
avg_60_forecastio_windBearing	0.0159723111776
avg_1440_lmse_calib_as_co	0.0149001557286
avg_720_lmse_scaled_sharpDust	0.0148211173957
day_of_year	0.0145567862081
avg_60_forecastio_pressure	0.0142569975814
daily_avg_sck_humidity	0.013849933762
avg_30_ws	0.0137791673751
daily_avg_forecastio_temperature	0.0136871069105

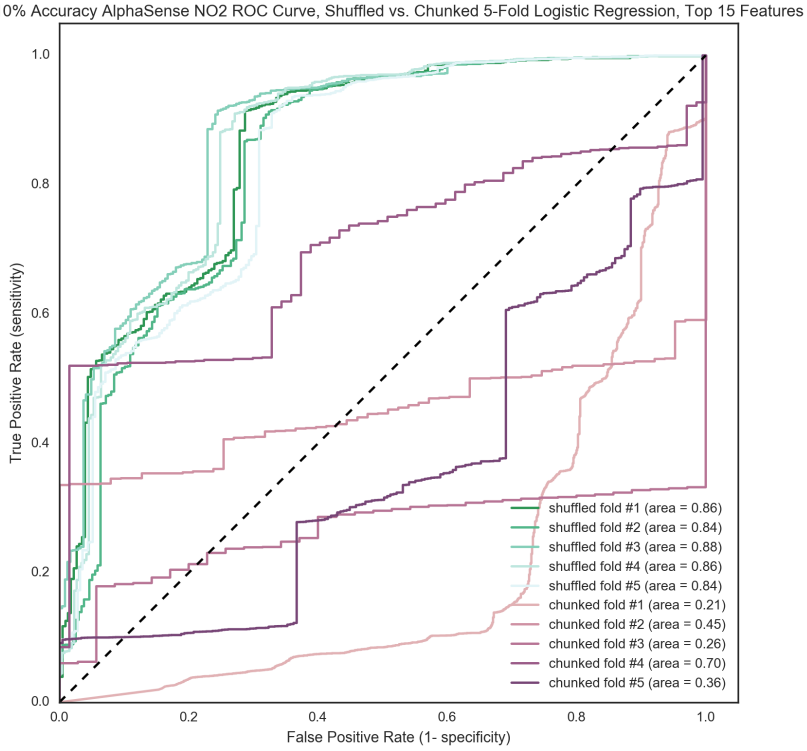


Figure 89: AlphaSense NO_2 ROC Using Top 15 Features

AlphaSense O3

Following are additional plots from the AlphaSense O3 test outlining the complete raw data and LMSE process, the accuracy with a tighter 7.5% threshold, a visualization of the prediction accuracy and confidence, and the top 15 random forest selected features for both sensors tested. Additionally, the ROC plots using just the top 15 features is included for both sensors.

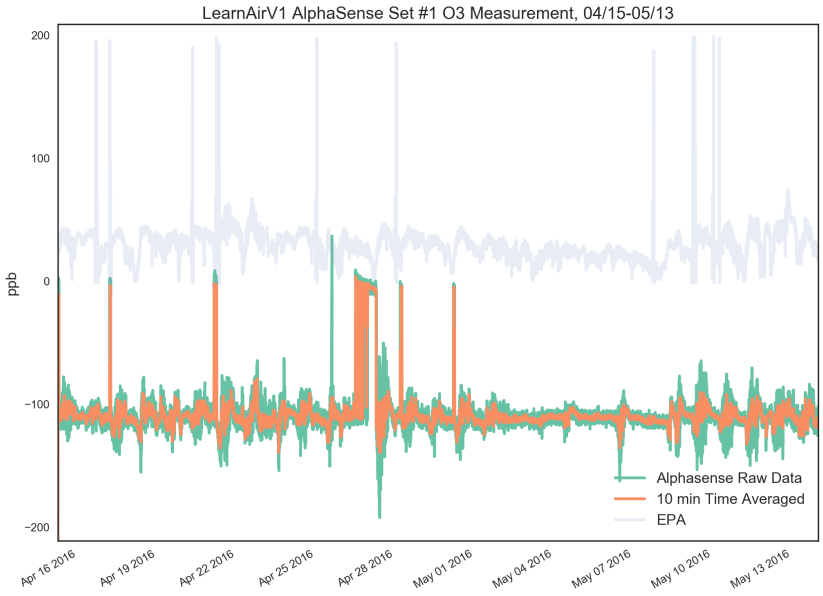


Figure 90: AlphaSense O3 Sensor 1 Raw Data

Table 34: Top 15 Features from Random Forest for O3 Sensor 1, used in Pruned Logistic Regression

Feature	Importance
lmse_calib_as_03	0.0388052728559
as_03	0.038535780326
alphaS1_work	0.0256714422814
avg_10_as_03	0.0143759967966
avg_10_lmse_calib_as_03	0.0143305118271
as_h2s	0.0131703607364
avg_720_bkcarbon	0.0131177282572
avg_60_bkcarbon	0.0128714271405
avg_1440_bkcarbon	0.0124171125826
min_since_plugged_in	0.012288250826
bkcarbon	0.0122005264086
avg_1440_lmse_scaled_sharpDust	0.0118815562888
avg_720_lmse_scaled_sharpDust	0.0116399996115
daily_avg_as_temperature	0.0116365039598
alphaS3_work	0.0115947138563

Table 35: Top 15 Features from Random Forest for O3 Sensor 2, used in Pruned Logistic Regression

Feature	Importance
avg_720_bkcarbon	0.0190323311421
avg_1440_bkcarbon	0.0186458328226
avg_1440_as_co	0.0185435347092
daily_avg_as_temperature	0.0177518437915
lmse_calib_as_03	0.0170424373503
daily_avg_forecastio_temperature	0.0170308459472
avg_60_bkcarbon	0.0167897682822
min_since_plugged$_in$	0.0166420124401
avg_60_forecastio_pressure	0.016620265085
bkcarbon	0.015766200638
daily_avg_forecastio_humidity	0.0147498428636
avg_60_forecastio_apparentTemperature	0.0140509238175
forecastio_pressure	0.013926105091
avg_60_forecastio_temperature$_c$	0.0136171116857
day_of_year	0.0133242379573

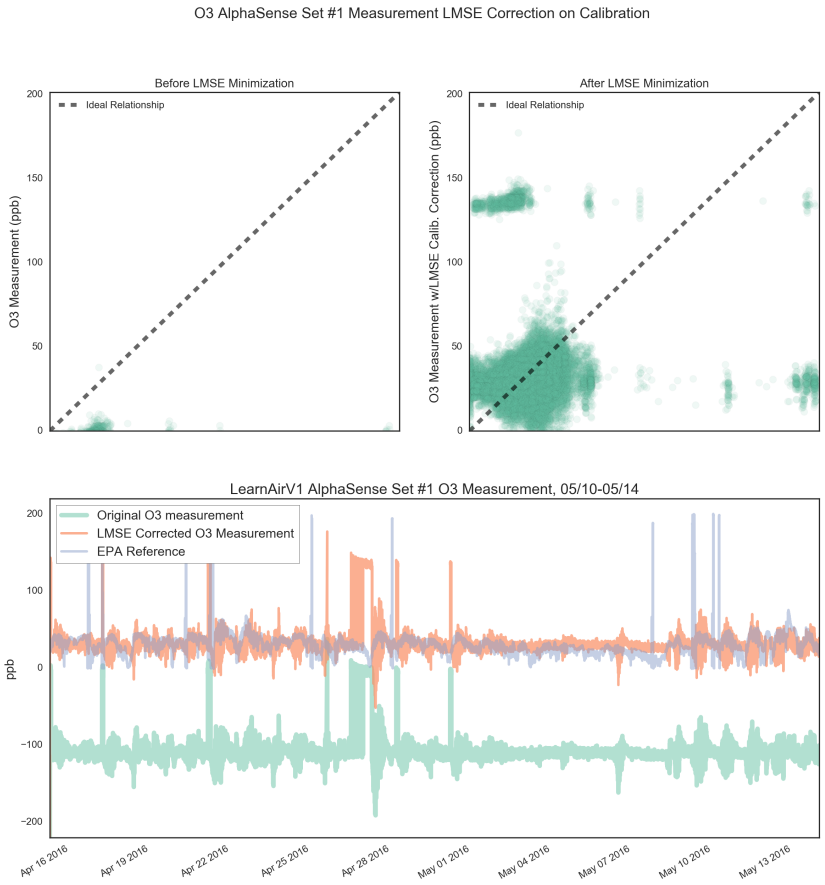


Figure 91: AlphaSense O3 Sensor 1 after LMSE Calibration

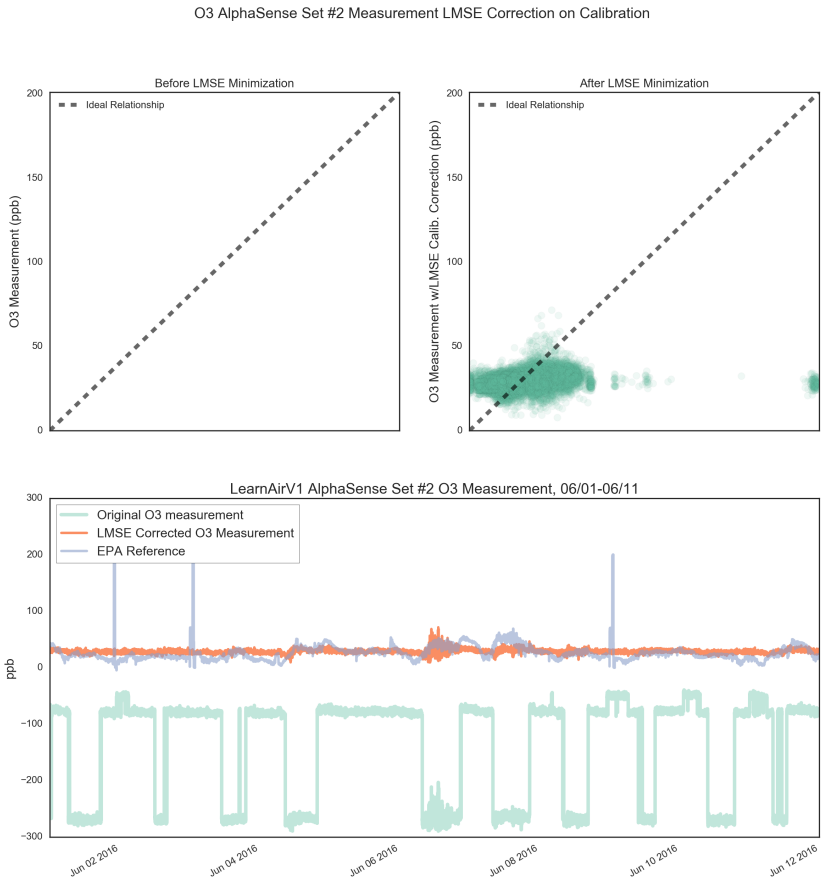


Figure 92: AlphaSense O3 Sensor 2 after LMSE Calibration

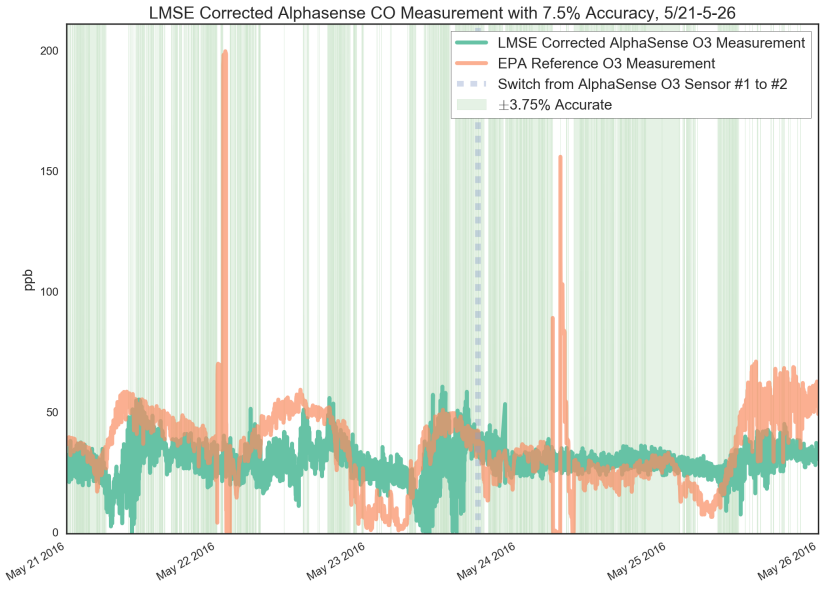


Figure 93: AlphaSense O3 Sensor 1 and 2 with 7.5% Accuracy Threshold

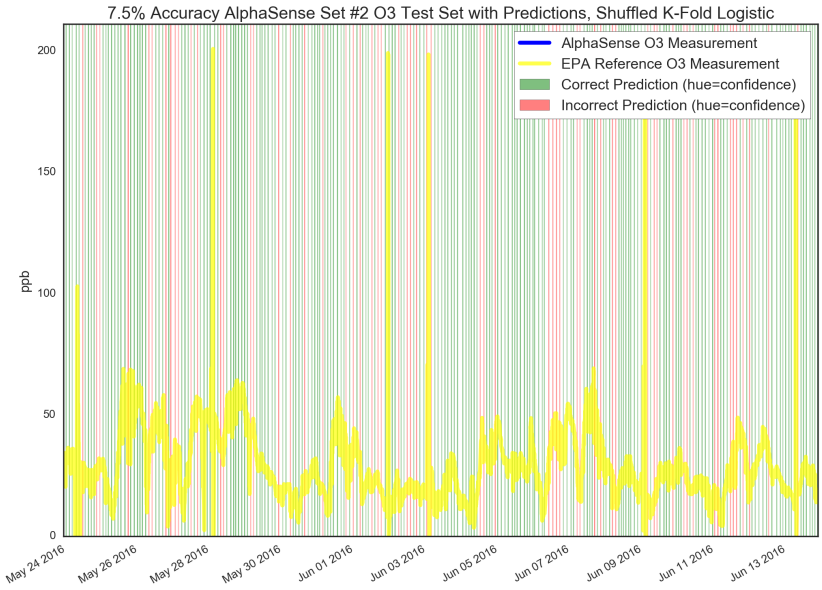


Figure 94: AlphaSense O3 Sensor 2 Prediction Accuracy

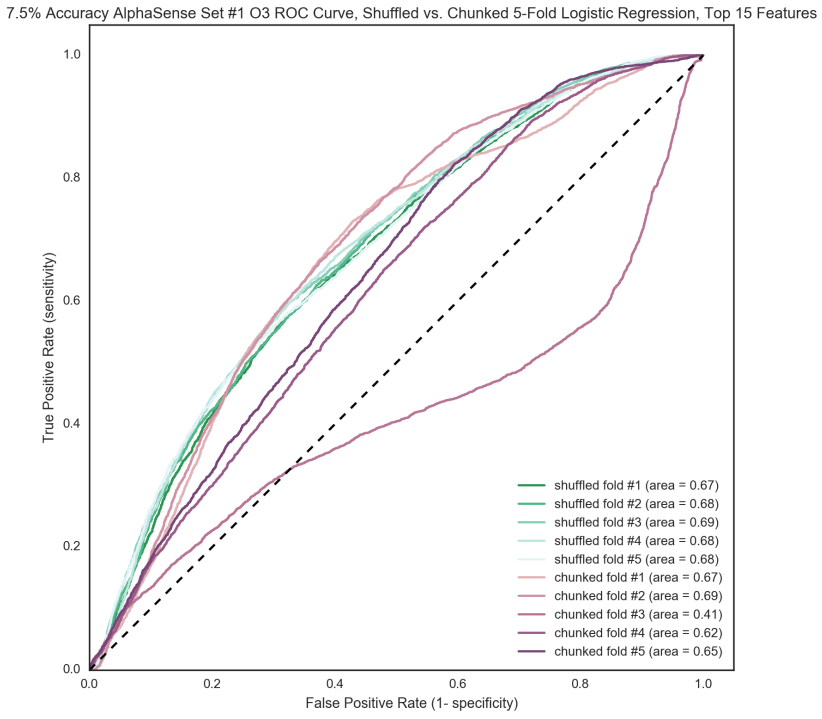


Figure 95: AlphaSense O3 Sensor 1 ROC Using Top 15 Features

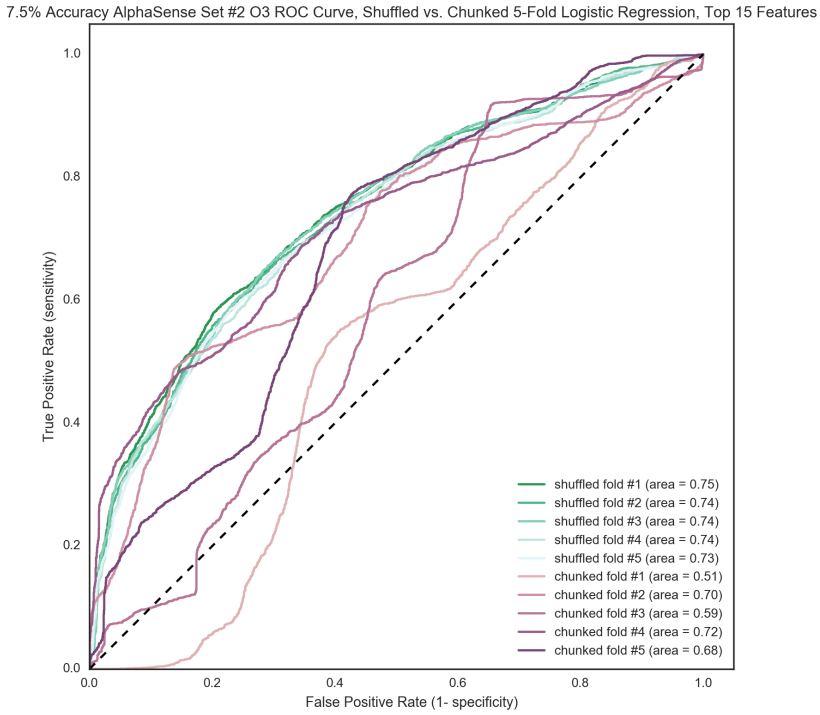


Figure 96: AlphaSense O3 Sensor 2 ROC Using Top 15 Features

Bibliography

[1] WHO News Release. 7 million premature deaths annually linked to air pollution., March 2014.

[2] WHO Report. Burden of disease from air pollution 2012, 2014.

[3] Giles F. Filley, Donald J. MacIntosh, and George W. Wright. Carbon monoxide uptake and pulmonary diffusing capacity in normal subjects at rest and during exercise. *Journal of Clinical Investigation*, 33(4):530, 1954.

[4] Elizabeth N. Allred et al. Short-term effects of carbon monoxide exposure on the exercise performance of subjects with coronary artery disease. *New England Journal of Medicine*, 321(21):1426–1432, 1989.

[5] Joel Schwartz et al. Traffic related pollution and heart rate variability in a panel of elderly subjects. *Thorax*, 60(6):455–461, 2005.

[6] Ron Williams et al. *Evaluation of Field-deployed Low Cost PM Sensors*. EPA Evaluation Report EPA/600/R-14/464, 2014.

[7] John V. Molenar. Theoretical analysis of pm2.5 mass measurements by nephelometry - #110. *Colorado State University Press*, 2005.

[8] Alfred H. Lowrey, Lance A. Wallace, et al. *Combustion Efficiency and Air Quality*, chapter 10: Concentrations of Combustion Particulates in Outdoor and Indoor Environments, pages 175–211. Plenum Press, 1995.

[9] Environmental Protection Agency. National air quality and

emissions trends report. 2014.

[10] *Particulate Matter in the UK*, chapter 5: Methods for Monitoring Particulate Concentration. Department for Environment, Food, and Rural Affairs, 1999.

[11] Anna Morpurgo, Federico Pedersini, and Alessandro Reina. A low-cost instrument for environmental particulate analysis based on optical scattering. In *2012 IEEE International*, 2012.

[12] Jayakarthigeyan Prabakar et al. Evaluation of low cost particulate matter sensor for indoor air quality measurement. *International Journal of Innovative Research in Science, Engineering, and Technology*, 4(2):366–369, 2015.

[13] Elena Austin et al. Laboratory evaluation of the shinyei ppd42ns low-cost particulate matter sensor. *PloS one*, 10(9):e0137789, 2015.

[14] Compact, Low Cost Particle Sensor. Roger L Unger, assignee. Patent US 8009290 B2. 30 Aug. 2011. Print.

[15] personal communication. Dr. Jesse Kroll and David Hagan, MIT Department of Civil and Environmental Engineering.

[16] Air Quality Sensor Performance Evaluation Center. Field evaluation alphasense opc-n2 sensor. Technical report, South Coast Air Quality Management District, 2015s.

[17] Alphasense Application Note. AAN 104: How electrochemical gas sensors work. Technical report, 2014.

[18] Susanne Steinle, Stefan Reis, and Clive Eric Sabel. Quantifying human exposure to air pollution—moving from static monitoring to spatio-temporally resolved personal exposure assessment. *Science of the Total Environment*, 443:184–193, 2013.

[19] David Hasenfratz et al. Pushing the spatio-temporal resolution limit of urban air pollution maps. In *2014 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2014.

[20] Sotiris Vardoulakis et al. Modelling air quality in street canyons: A review. *Atmospheric Environment*, 37(2):155–182, 2003.

[21] Shaodong Xie et al. Spatial distribution of traffic-related pollutant concentrations in street canyons. *Atmospheric Environment*, 37(23):3213–3224, 2003.

[22] Agata Rakowska et al. Impact of traffic volume and composition on the air quality and pedestrian exposure in urban street canyon. *Atmospheric Environment*, 98:260–270, 2014.

[23] Soni Kaur, Mark J. Nieuwenhuijsen, and Roy N. Colvile. Fine particulate matter and carbon monoxide exposure concentrations in urban street transport microenvironments. *Atmospheric Environment*, 41(23):4781–4810, 2007.

[24] Stuart Batterman, Sarah Chambliss, and Vlad Isakov. Spatial resolution requirements for traffic-related air pollutant exposure evaluations. *Atmospheric Environment*, 94:518–528, 2014.

[25] Colby Adams, Philip Riggs, and John Volckens. Development of a method for personal, spatiotemporal exposure assessment. *Journal of Environmental Monitoring*, 11(7):1331–1339, 2009.

[26] Gerard Hoek et al. A review of land-use regression models to assess spatial variation of outdoor air pollution. *Atmospheric Environment*, 42(33):7561–7578, 2008.

[27] H. K. Lai et al. Personal exposures and microenvironment concentrations of pm 2.5, voc, no 2 and co in oxford, uk. *Atmospheric Environment*, 38(37):6399–6410, 2004.

[28] Juana Maria Delgado-Saborit. Use of real-time sensors to characterise human exposures to combustion related pollutants. *Journal of Environmental Monitoring*, 14(7):1824–1837, 2012.

[29] Anju Goel and Prashant Kumar. A review of fundamental drivers governing the emissions, dispersion and exposure to vehicle-emitted nanoparticles at signalised traffic intersections. *Atmospheric Environment*, 97:316–331, 2014.

[30] Yungang Wang et al. Roadside measurements of ultrafine particles at a busy urban intersection. *Journal of the Air Waste Management Association*, 58(11):1449–1457, 2008.

[31] Yun Cheng, Xiucheng Li, et al. Aircloud: a cloud-based air-quality monitoring system for everyone. *SENSYS*, 2014.

[32] Vijay Sivaraman, James Carrapetta, et al. Hazewatch: A participatory sensor system for monitoring air pollution in sydney. *Eighth IEEE Workshop on Practical Issues in Building Sensor Network Applications*, pages 56–64, 2013.

[33] Prabal Dutta, Paul M. Aoki, et al. Common sense: Participatory urban sensing using a network of handheld air quality monitors. *Proceedings of the 7th ACM conference on embedded networked sensor systems*, 2009.

[34] David Hasenfratz, Olga Saukh, et al. Opensense zurich: A system for monitoring air pollution. *Presented at the Nano-Tera Annual Plenary Meeting in Bern*, 2011.

[35] Smartcitizen sensor. `https://smartcitizen.me`, 2013.

[36] Matt Burgess. London gets a flock of air pollution monitoring pigeons. Wired News, `http://www.wired.co.uk/article/london-pollution-pigeon-air-patrol`, 2016.

[37] Natasha Khan. Are cheap sensors and concerned citizens leading to a shift in air monitoring? PublicSource News, `http://publicsource.org/investigations/are-cheap-sensors-and-concerned-citizens-leading-shift-air-monitoring`, 2015.

[38] Air quality sensor performance evaluation center. Technical report, South Coast Air Quality Management District, 2015.

[39] Spencer Russell. Chainapi repository. `https://github.com/ResEnv/chain-api`, 2014.

[40] Spencer Russell and Joseph Paradiso. Hypermedia apis for sensor data: A pragmatic approach to the web of things. *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 30–39, 2014.

[41] Responsive Environments, MIT Media Lab. Tidmarsh project. `http://tidmarsh.media.mit.edu/`, 2013.

[42] Alexandra Moraru, Marko Pesko, Maria Porcius, Carolina Fortuna, and Dunja Mladenic. Using machine learning on sensor data. *Journal of Computing and Information Technology*, 18(4):341–347, 2010.

[43] Sajjad Ahmad, Ajay Kalra, and Haroon Stephen. Estimating soil moisture using remote sensing data: A machine learning approach. *Advances in Water Resources*, 33(1):69–80, 2010.

[44] Matt Smith, Charles Castello, and Joshua New. Machine learning techniques applied to sensor data correction in building technologies. *12th International Conference on Machine Learning and Applications (ICMLA)*, pages 305–308, 2013.

[45] M. Strano and B.M. Colosimo. Logistic regression analysis for experimental determination of forming limit diagrams. *International Journal of Machine Tools and Manufacture*, 46(6):673–682, 2006.

[46] Safecast. `http://www.safecast.org`, 2011.

[47] Aclima. `http://aclima.io`, 2010.

[48] Copenhagen wheel. `http://superpedestrian.com`, 2009.

[49] David Hasenfratz et al. Participatory air pollution monitoring using smartphones. *Mobile Sensing*, 2012.

[50] David Hasenfratz, Olga Saukh, and Lothar Thiele. On-the-fly calibration of low-cost gas sensors. *Wireless Sensor Networks*, pages 228–244, 2012.

[51] Olga Saukh, David Hasenfratz, and Lothar Thiele. Reducing multi-hop calibration errors in large-scale mobile sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, 2015.

[52] Brian Mayton, Gershon Dublon, Sebastian Palacios, and Joseph Paradiso. Truss: Tracking risk with ubiquitous smart sensing. *IEEE Sensors*, pages 1–4, 2012.

[53] Clairity. `http://clairity.mit.edu/site/html/home.html`, 2014.

[54] Tricorder project. `http://www.tricorderproject.org/`, 2007.

[55] Propeller health. `http://www.propellerhealth.com`, 2010.

[56] Upod project. `http://mobilesensingtechnology.com/`, 2016.

[57] Giuseppe Anastasi, Paolo Brushci, et al. 'u-sense', a cooperative sensing system for monitoring air quality in urban areas. *ERCIM News*, 2014.

[58] Airbeam project. http://aircasting.org/, 2014.

[59] Elm project. http://elm.perkinelmer.com/map/, 2013.

[60] Breezometer project. https://breezometer.com/, 2015.

[61] Cleanspace project. https://our.clean.space/, 2016.

[62] Clarity pm2.5 sensor. http://clarity.io, 2014.

[63] Tzoa enviro-tracker. http://www.tzoa.com, 2014.

[64] uhoo indoor sensor. https://uhooair.com/s, 2016.

[65] Air quality egg project. http://airqualityegg.com/, 2012.

[66] Speck sensor. https://www.specksensor.com/, 2015.

[67] Atmotube. http://atmotube.com/, 2015.

[68] Airbox lab. http://foobot.io/, 2013.

[69] Citizenair.io. http://www.citoyenscapteurs.net/2014/03/31/citizenair-io/, 2014.

[70] Public lab. https://publiclab.org/, 2010.

[71] Django framework. https://www.djangoproject.com/, 2005.

[72] Apache hadoop. http://hadoop.apache.org/, 2011.

[73] RDF Working Group. Rdf 1.1 schema, concepts and abstract syntax. Technical report, World Wide Web Consortium, 2014.

[74] JSON-LD. http://json-ld.org/, 2010.

[75] HAL - hypertext application language. http://stateless.co/hal_specification.html, 2013.

[76] Iotivity standard. https://www.iotivity.org/, 2015.

[77] Alljoyn framework. `https://allseenalliance.org/framework`, 2011.

[78] Evan Lynch and Joseph Paradiso. Sensorchimes: Musical mapping for sensor networks. *New Interfaces for Musical Expression*, 2016.

[79] QianSheng Li, Gershon Dublon, Brian Mayton, and Joseph A. Paradiso. Marshvis: Visualizing real-time and historical ecological data from a wireless sensor network. *IEEE VIS*, 2015.

[80] Cheng Liu et al. A directional anemometer based on mems differential pressure sensors. *9th International Conference on Nano/Micro Engineered and Molecular Systems (NEMS)*, pages 517–520, 2014.

[81] Steven G. Brown, Paul T. Roberts, et al. Characterization of 2001 ozone event-triggered voc and carbonyl samples in houston. Technical report, Sonoma Technology, Inc. Presented at TCEQ State of the Science Meeting, 2002.

[82] Eduardo P. Olaguer. Near-source air quality impacts of large olefin flares. *Journal of the Air Waste Management Association*, 62(8):978–988, 2012.

[83] C. Treviño and F. Méndez. Simplified model for the prediction of ozone generation in polluted urban areas with continuous precursor species emissions. *Atmospheric Environment*, 33(7):1103 – 1110, 1999.