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
Thesis Supervisor

Accepted by .....
Pattie Maes
Academic Head

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

—This is the OLD Abstract from my proposal, and is not up-to-date. It will be updated shortly—

Air pollution is responsible for 1/8 of deaths around the world. Studies have conclusively proven that the status quo of sparse, fixed sensors cannot accurately capture personal exposure levels of nearby populations. Especially in urban landscapes, pollutant concentrations can swing wildly over just a few seconds or a few meters. Increasingly, mobile solutions are being tested for mapping cities and measuring personal exposure. Affordable and portable sensing technology has just reached the accuracy necessary for meaningful data collection. Unfortunately, current systems are not robust at dealing with rapid pollution variation, dynamic airflow, or changes in temperature, humidity, and pressure.

For this work, we will explore novel techniques for addressing the shortcomings of the two types of sensors most commonly used for mobile sensing– Particulate Sensing and Electrochemical Gas Sensing. We will prototype several new designs in an attempt to mitigate issues with airflow, spatiotemporal pollutant variation, temperature, humidity, and pressure. We will test and characterize our designs (1) in controlled dynamics and airflow conditions, (2) in real urban settings, and (3) in comparison with higher quality reference sensors, with a particular focus on how they perform under various speeds of travel (walking, running, biking, and driving). Combined with a smartphone, we believe we can produce a reliable and affordable sensor platform.

Ultimately, we want to empower citizen groups to understand and combat air pollution in their community. For people who are actively trying to mitigate their exposure to harmful pollutants, we want to help them understand their exposure, their risk, and whether their actions are working. For communities where no data exists, we want to empower affordable, distributed, and verifiably accurate data collection that can serve as a basis for civic discourse. We believe these are achievable goals with the current technology, as long as these designs are truly built and tested for the mobile context.

Thesis Supervisor: Joseph A. Paradiso

Title: Professor

LearnAir: toward Intelligent, Personal Air Quality Monitoring

David B. Ramsay

The following people served as readers for this thesis:

Thesis Reader .....

Steven Hamburg
Chief Scientist
Environmental Defense Fund

Thesis Reader .....

Ethan Zuckerman
Professor
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. 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 **EPA Dave**, **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, 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, Will, Chetan, 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 Huang** - 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 reexamine 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

1. Introduction 19

2. Background and Motivation 23
- *Air Monitoring* 23
- *Sensor Networks* 30
- *Motivation* 33

3. Related Work 37
- *Air Quality* 37
- *Data Sharing Solutions* 39

4. Overview of Design and Contributions 43
- *Machine Learning Validation* 44
- *ChainAPI Instance and Tools* 45
- *A Provocative Example* 46

5. Hardware Design and Analysis 49
- *LearnAir Version 1* 50
- *MassDEP Site* 53
- *LearnAir Version 2* 56
- *LearnAir Version 3* 59
- *Hardware Comparison and Analysis* 61

6. ChainAPI for Air Quality 67
- *A New Ontology for Air Quality* 69

Traversing ChainAPI 74
ChainAPI Tools for Scalable, Automatic Data Analysis 82
Summary 93

7. Data Analysis and Machine Learning 95
Test Conditions and Data Collection Summary 96
Overview of Data Pre-Processing and ML Strategy 98
Machine Learning Features 101
SmartCitizen CO 107
SmartCitizen NO2 114
Sharp Dust Sensor 121
AlphaSense CO 129
AlphaSense NO2 141
AlphaSense O3 149

8. Conclusions and Future Work 163
Applications 163
Insights 163
Future Work 164
Summary 164

Appendix A - Notes on Project Selection and Prior Work 167

Appendix B - Hardware and Firmware 169
Schematics 169
Firmware 174
Hardware Analysis 174

Appendix C - ChainAPI Code 177

Appendix D - Machine Learning 179
Test Conditions and Data Summary 179
Data Pre-Processing 183
Features 183
SmartCitizen CO 183

SmartCitizen NO2 183
Sharp Dust Sensor 183
AlphaSense CO 183
AlphaSense NO2 183
AlphaSense O3 183

List of Figures

1 AlphaSense Opitcal PM2.5 Sensors. 25
2 AlphaSense Electrochemical Gas Sensors. 27

3 LearnAir Sensor installed at MassDEP site, opened. 50
4 LearnAir Sensor installed at MassDEP site. 51
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. 51
6 A picture of the Roxbury MassDEP measurement site where the LearnAir sensor was installed. 53
7 LearnAir Sensor (box on left) installation next to MassDEP inlet (top of pole on right). 55
8 Main and daughter boards of learnAir V2.0. 56
9 Second revision, Atmel based learnAir main board mated with the AlphaSense sensor frontend. 57
10 Final design of the portable system. 58
11 Layouts for revisions 2.0 and 3.0 of the learnAir board. 59
12 Third revision, STM32L152-based learnAir main board next to the AlphaSense sensors. 60
13 Humidity Comparison of SmartCitizen (orange) and ForecastIO (green) over 4 days 61
14 Humidity Comparison, SmartCitizen and ForecastIO 62
15 Alphasense Raw Temperature Data (green) with 15-minute averaging (orange) 62
16 Temperature Close up 63
17 Temperature Comparison, SmartCitizen and ForecastIO 63
18 A picture of a simple laminar flow test setup for rough wind directivity characterization. 64
19 Wind Directivity Polar Pattern, Axis = 1 64
20 Wind Directivity Polar Pattern, Axis = 2 64
21 Wind Speed Measurement with 10% Accuracy, Zoomed 65
22 Windspeed Error vs Wind Direction 66

23 Summary of New ChainAPI Infrastructure. 67

24 Weather during Test Period 96
25 Temperature and Humidity during Test Period 97
26 All ML features plotted with WEKA Tool 104
27 Temperature during Test Period 105
28 Humidity during Test Period 105
29 Temperature Derivative Feature Creation 106
30 Humidity Derivative Feature Creation 106
31 SmartCitizen CO Raw Data 107
32 SmartCitizen CO after LMSE Calibration 108
33 SmartCitizen CO with 7.5% Accuracy Threshold 109
34 SmartCitizen CO with 5% Accuracy Threshold 109
35 SmartCitizen CO ROC Curve 111
36 SmartCitizen CO ROC Using Top 15 Features 112
37 SmartCitizen CO Prediction Accuracy 112
38 SmartCitizen NO2 Raw Data 114
39 SmartCitizen NO2 after LMSE Calibration 115
40 SmartCitizen NO2 with 10% Accuracy Threshold 115
41 SmartCitizen NO2 with 4% Accuracy Threshold 116
42 SmartCitizen NO2 ROC Curve 117
43 SmartCitizen NO2 ROC Using Top 15 Features 118
44 SmartCitizen NO2 Prediction Accuracy 119
45 Sharp Raw Particulate Data 121
46 Sharp Particulate LMSE Calibration 122
47 2 day Average Sharp Particulate LMSE Calibration 123
48 Sharp Particulate with 30% Accuracy Threshold 123
49 Sharp Particulate ROC Curve 124
50 Sharp Particulate ROC Using Top 15 Features 126
51 Sharp Particulate Prediction Accuracy 127
52 AlphaSense CO Sensor 1 Raw Data 129
53 AlphaSense CO Sensor 1 Raw Data Zoomed 130
54 AlphaSense CO Sensor 1 after LMSE Calibration 131
55 AlphaSense CO Sensor 2 after LMSE Calibration 132
56 AlphaSense CO Sensor 1 and 2 with 7.5% Accuracy Threshold 132
57 AlphaSense CO Sensor 1 and 2 with 5% Accuracy Threshold 133
58 AlphaSense CO Sensor 1 ROC Curve 133
59 AlphaSense CO Sensor 1 ROC Using Top 15 Features 135
60 AlphaSense CO Sensor 1 Prediction Accuracy 136
61 AlphaSense CO Sensor 2 ROC Curve 136
62 AlphaSense CO Sensor 2 ROC Using Top 15 Features 139
63 AlphaSense CO Sensor 2 Prediction Accuracy 140
64 AlphaSense NO2 Raw Data 141
65 AlphaSense NO2 Raw Data Zoomed 142

66 AlphaSense NO2 after LMSE Calibration 143
67 AlphaSense NO2 with 10% Accuracy Threshold 143
68 AlphaSense NO2 with 4% Accuracy Threshold 144
69 AlphaSense NO2 ROC Curve 146
70 AlphaSense NO2 ROC Using Top 15 Features 147
71 AlphaSense NO2 Prediction Accuracy 147
72 AlphaSense O3 Sensor 1 Raw Data 150
73 AlphaSense O3 Sensor 1 Raw Data Zoomed 151
74 AlphaSense O3 Sensor 1 after LMSE Calibration 151
75 AlphaSense O3 Sensor 2 after LMSE Calibration 152
76 AlphaSense O3 Sensor 1 and 2 with 7.5% Accuracy Threshold 152
77 AlphaSense O3 Sensor 1 and 2 with 5% Accuracy Threshold 153
78 AlphaSense O3 Sensor 1 ROC Curve 155
79 AlphaSense O3 Sensor 1 ROC Using Top 15 Features 156
80 AlphaSense O3 Sensor 1 Prediction Accuracy 156
81 AlphaSense O3 Sensor 2 ROC Curve 159
82 AlphaSense O3 Sensor 2 ROC Using Top 15 Features 160

84 Original Concept #1 167
85 Original Concept #2 168

86 Wind Speed Measurement with 10% Accuracy 175
87 Wind Direction with 10% Accuracy WindSpeed Measurements Denoted 176

88 Precipitation Intensity during Test Period 179
89 Ambient Pressure during Test Period 180
90 Cloud Cover during Test Period 180
91 Dew during Test Period 181
92 Lux during Test Period 181
93 Lux during Test Period 182

List of Tables

1 Machine Learning Features used to Predict Sensor Accuracy 103
2 Error Rates for Predicting SmartCitizen CO Accuracy with Logistic Regression 109
3 Average SmartCitizen CO Confusion Matrix w/Shuffled K-Fold 110
4 Top 15 Features from Random Forest for SmartCitizen CO, used in Pruned Logistic Regression 111
5 Top Features for Predicting SmartCitizen CO 113
6 Error Rates for Predicting SmartCitizen NO_2 Accuracy with Logistic Regression 116
7 Average SmartCitizen NO_2 Confusion Matrix w/Shuffled K-Fold 117
8 Top 15 Features from Random Forest for SmartCitizen NO_2, used in Pruned Logistic Regression 118
9 Top Features for Predicting SmartCitizen NO_2 120
10 Error Rates for Predicting Sharp Accuracy with Logistic Regression 124
11 Average Sharp Particulate Confusion Matrix w/Shuffled K-Fold 125
12 Top 15 Features from Random Forest for Sharp Sensor, used in Pruned Logistic Regression 126
13 Top Features for Predicting Sharp Particulate 128
14 Error Rates for Predicting CO Sensor 1 Accuracy with Logistic Regression 131
15 Average AlphaSense CO Sensor 1 Confusion Matrix w/Shuffled K-Fold 134
16 Top 15 Features from Random Forest for CO Sensor 1, used in Pruned Logistic Regression 135
17 Top Features for Predicting AlphaSense CO Sensor 1 137
18 Error Rates for Predicting CO Sensor 2 Accuracy with Logistic Regression 137
19 Average AlphaSense CO Sensor 2 Confusion Matrix w/Shuffled K-Fold 138
20 Top 15 Features from Random Forest for CO Sensor 2, used in Pruned Logistic Regression 139

22 Error Rates for Predicting AlphaSense NO_2 Accuracy with Logistic Regression 144
23 Average AlphaSense NO_2 Confusion Matrix w/Shuffled K-Fold 145
24 Top 15 Features from Random Forest for AlphaSense NO_2, used in Pruned Logistic Regression 146
25 Top Features for Predicting AlphaSense NO_2 148
26 Error Rates for Predicting O_3 Sensor 1 Accuracy with Logistic Regression 153
27 AlphaSense O_3 Sensor 1 Confusion Matrix w/Shuffled K-Fold 154
28 Top 15 Features from Random Forest for O_3 Sensor 1, used in Pruned Logistic Regression 155
29 Top Features for Predicting AlphaSense O_3 Sensor 1 157
30 Error Rates for Predicting O_3 Sensor 2 Accuracy with Logistic Regression 157
31 AlphaSense O_3 Sensor 2 Confusion Matrix w/Shuffled K-Fold 158
32 Top 15 Features from Random Forest for O_3 Sensor 2, used in Pruned Logistic Regression 159
33 Top Features for Predicting AlphaSense O_3 Sensor 2 161

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. 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. 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 this relationship can undermine and obfuscate the health risks facing our global community.

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 has proven reliable under real-world conditions. 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 problem on larger scales. In many cities, politicial 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 makes it easy for smart city initiatives to find themselves with poor quality solutions. At best this is a waste of time and money; at worst, it may cause unneccessary 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 now playing from behind. Long-term, a saturated market of unreliable technology could lead to mistrust and disillusionment with this issue and its proponents.

While well-intentioned organizations fight to figure out how to interact and guide community sensing intiatives to promote accuracy, awareness, and involvement, they also have a lot to gain from distributed, reliably-measured personal data. A break through in personal sensing could unlock amazing insight into pollution modeling and epidemiology. How to store, interact with, characterize, and use this data is an open question in the environemental sensing world. Even simple sharing of data amongst trustworthy organizations and researchers– without the confounding factor of unknown data quality– is an open and actively-debated issue. Differences in data labeling and data collection methodology make it difficult to agree upon a set of defintions and data collection standards.

There are many serious issues and much unrealized potential– for individuals, policy-makers, and organizations– that arise from the lack of trustworthy, affordable air quality monitoring solutions. 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. There is also no agreed-upon standard for characterizing these device in real-world conditions, even with demonstrable success. 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 sensors, our goal is to simply *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. Unfortunately there is no systemic 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 intelligently address their exposure concerns. It will also empower researchers and organizations to interact with data from historically untrustworthy sources. Furthermore, the algorithms underlying the learnAir system will 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 backend. 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 backend design has powerful implications for the air quality community. It not only points to a principles for sharing air quality data, but also stakes a claim in how researchers and engineers might start to think about and interact with any distributed data set where quality is a significant variable.

The LearnAir hardware is a portable, 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 check to see when it is correct by comparing itself against the reference. When it sees an incorrect reading, it analyzes the weather and other ambient conditions to see if there are any patterns that correlate with its inaccuracy. It draws similar comparisons with patterns that suggest it is making accurate readings.

Thus, every measurement learnAir makes comes with a prediction as to whether it is correct, as well as a probability that it has guessed its correctness accurately. This information is based on a shared data resource, which houses an constantly updating sensor model that takes into account all measurements ever reported to the system by a sensor of that type. In order to build this device and the supporting infrastructure, and in order to validate that a system of this type is useful, we need to answer two preliminary, important questions:

(1) How well can machine learning algorithms predict the accuracy of a complex, unreliable sensor given the some information about the sensor's environment, and what is the best way to apply machine learning to this problem? It is a fundamental research question whether or not machine learning will give meaningful results for this use case. Answering this question successfully has implications beyond air quality sensor networks, as an exploration of new topologies for high/low quality sensor interaction and system design.

(2) Secondly, how do we support an intelligent, scalable, open data structure that supports different systems of variable quality? 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.

Answering these two questions, as well as using their results to build a novel, deployable system, are the three main contributions of this thesis. Fundamentally, we want to test whether machine learning can give us a more nuanced understanding of when and how cheap sensors succeed, so we can use their data and elevate their trustworthiness. We want to build a scalable database where sensor data of any quality co-exists seamlessly, which supports scalable learning algorithms and easy-to-use, curated data. And finally, we want to demonstrate a system that takes advantages of these techniques, provokes a dialog in the citizen sensing community, and puts these new techniques 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 represents months of distilled advice from industry experts and thought leaders, who greatly informed the direction of this work.

Air Monitoring

Air Pollutants

Despite the complexities of behind sparce air quality measurement and individual health, there is a plethora of evidence linking small particulate matter and other pollutants to serious negative health effects.

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. Nanoparticles 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 there is precipitation.

Particulate size distribution is generally viewed as the sum of n log-normal distributions. Nucleation by-products from engine combustion drives a positively-skewed, log-normal distribution centered at a few hundred nanometers. 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. 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, Sulfer Dioxide, and Ozone.

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 is almost a mute point outside of old house paint. 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 of high exposure.

Sulfur Dioxide is the result of fossil fuel combustion, 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 ppb average per hour for SO2, 100 ppb average per hour for NO2, 70 ppb average for 8 hours for Ozone, 9 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.

Sensor Technologies for Particulate Matter

Figure 1: AlphaSense Opitcal PM2.5 Sensors.

There are many sensor modalities for pollution monitoring, with a handful separating themselves as the standard references. For particulate, high quality, large installations will usually either have a Beta Attenuation Monitor (BAM) or a Tapered Element Oscillating Microbalance (TEOM) sensor for particulate sensing. 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 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.

Optical methods for particulate sensing are particularly important in the mobile context. 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 mixture 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, airflow through the device is not tightly controlled (it may be driven by convection with a small heating element, but it is never precise), 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 simply count pulses as the beam breaks– with a known, precisely controlled airflow, 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

calculates particle size by looking at intensity of the scattered light on the photodiode (a nonlinear monotonic function with particle size). In this way the duration of a particle in front of the beam can be used independently to verify flow rate, and address small errors in its control. In all of these designs, larger particles have a non-trivial deposition rate, which further constrains the geometry and flow through a device. Expensive 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 are included on all professional level equipment. These inlets protect the device from wind, create predictable airflow, and typically include particle size selectivity. Most commonly, size selectivity is achieved using inertial techniques (i.e. impaction or cyclone filtering). They are generally mounted in an upright position due to particle deposition (so gravity works with the inlet), and they are sometimes heated to evaporate fog.

For cheap, small, or mobile applications, optical sensing has been the dominant modality. On the high end (>$15k), handheld systems like the Grimm Enviro 11E have nice inlet and 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 clearly do not offer a viable option for distributed or personal applications.

On the cheaper end of the spectrum, many sensors exist. Unfortunately, independent tests have shown that none of these hold up in dynamic, real-world situations. One in particular worth noting is the Shinyei PPD42NS, a $10 sensor that is widely used and cited as providing high quality results. It has a small heating element for inducing convective flow, and coarse optics. Unfortunately, its reputation is based on very controlled conditions. In outdoor tests, the Shinyei provided erroneous results, demonstrating enormous cross-sensitivity to variations in airflow, temperature, and humidity. The lack of clarity around the useful application space of cheap optical sensors is an unfortunate common misconception, and demonstrates the importance of testing these sensors in realistic mobile contexts.

The $300 Dylos seems to be the cheapest optical sensor that stands

up reasonably well in independent tests. It uses an IR laser and has a much larger and more sophisticated flow design than its cheaper counterparts.

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 has not released a full characterization of their design, but preliminary testing in environmental sensing groups at MIT 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.

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 infering 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 method.

Figure 2: AlphaSense Electrochemical Gas Sensors.

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). 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.

Measurement Strategies and Complications

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

While these stations are highly accurate, studies have shown they either chronically underreport or have no correlation with the personal exposure of the citizens living near them. 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 to map and model a city using a small number of medium quality, mobile sensors have started to emerge. 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.

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 common phenomenon is called the 'urban canyon'– a street with two tall buildings on either side that creates several interacting, swirling vortices.

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. 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. Different corners of the same intersection can also vary tremendously. Even walking roadside vs. building-side on the same sidewalk has been linked to significant differences in pollution exposure level.

This is all to say that spatial variation is extremely high. Concentrations can change drastically over just a few meters. For true personal exposure monitoring, it is regarded as best practice to sample air within 30cm of the mouth and nose. While some of these spatial phenomena may fit an urban canyon model, and some may be modeled accurately with standard dispersion, few real-world scenarios are predictable with any single technique (especially to within a few meters). It is extremely difficult to predict real spatial variation without direct measurement.

Temporal variation is equally difficult to monitor. Studies at traffic intersections have shown that regular, tenfold increases in pollutant concentration occur over just one second. This staggering variation is averaged out even with the some of the best 'real-time' techniques– fifteen second integration would miss the entire event. Peak exposure levels may have important health implications, and in most cases transient events like this account for the majority of urban exposure.

Given the tremendous spatiotemporal variation in pollution, wear-

able air quality monitoring is clearly the only viable path to accurate personal exposure data. Fixed and mobile sensors, even with state-of-the-art predictive resolution, can't capture the detail required to estimate individual exposure (especially considering the accepted standard of 30 cm from the mouth and nose for accuracy).

With enough adoption, it is not difficult to image improving the collection and prediction of city-wide pollution mapping traditionally associated with fixed sensor installations. Eventually, wearable sensor data may even enable accurate path-based personal exposure modeling (statistical relevance on the order of meters and minutes), since the data is collected from varied users in the real microenvironments that dominate their exposure. Accounting for this otherwise highly specific spatiotemporal resolution would be difficult with any alternative method (day and 100m2 resolution is the best we see with predictive models right now). As mobile sensing 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. In most cases, these publications focus on system design, and produce thought-provoking work on the user-interface. In cases where technologists explore new sensor design, it is rare they achieve compelling improvements. The past 20 year 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 fundamentally new insight. Significant effort has brought us to current level of sophistication, and any incremental progress at the sensor level requires an equally sophisticated understanding.

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 cal-

ibration as they pass higher-quality fixed sensors. 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 launching on regular basis. Unfortunately, most devices do not stand up to independent scrutiny, and rarely do they offer meaningful improvements over the status quo. None of these devices has succeeded at sustaining momentum. 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). The constant barrage of 'new' monitoring devices– without accountability, without rigorous data-collection, and without real-world use-cases– saturates and dilutes consumer response to 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 reasonable quality. London (quite publicly) recently released a network of GPS-tracked, tweeting pidgeons with NO_2 sensor backpacks– while driven by a marketing firm as a (very successful) publicity campaign, it is a safe bet that data quality assurance is low.

The EPA publicly states that distributed, cheap sensing technology will be a cornerstone of their future success. As part of the effort to engage with active citizens and communities, the EPA measures and publishes data about low cost consumer devices. 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. 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), but no standard procedure has been agreed upon.

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 nice 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 form to address these problems, but the issues are still unresolved.

ChainAPI offers a thin, HAL- and JSON-based hypermedia solution for creating a distributed, browsable data resource. 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 ecological

installation in Southern Massachussetts. It serves as the backbone for many interesting data visualizations, audio compositions, 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). One notable research project used HVAC sensor data, both analytically and redundantly, to predict and verify whether other sensors in the network were working properly and automatically replace unreliable data.

Generally, all of 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. It can be applied as a binary classifier (i.e., 'Is this sensor failing?'), as well as give a probability for each outcome ('yes' or 'no').

Motivation

Air pollution is a health risk for people around the world. While standard measurement techniques are highly accurate, they are also extremely expensive and stationary. These stationary sensors fall short of capturing 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 exacting personal monitoring, wearable, mobile sensing is the

only answer. Sensors do exist that can measure personal exposure, but they are either thousands of dollars or they are cheap and inaccurate. While elusive, portable, 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 their reference level counterparts is truly remarkable. Cheap sensors are starting to mirror the core principles of instrument-grade devices. Unforunately, there are systemic failures in cheap systems that simply 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 would push a cheap, portable sensor out of its category. Electrochemical gas sensors require a clean, precision doping process, a statistically minimal exposed surface area, and compensation for fow 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 survey– 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).

The second take-away is that, excitingly, the core physics underlying cheap commercial sensors are approaching a very high quality. This suggests that the sensors– when they fail to provide accurate readings– are likely doing so because of systemic, predictable failures. If that was one single 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 underlying robustness.

The belief that cheap sensors are now entering a quality regime where failure is more predictable leads us to machine learning as a potentially powerful mechanism to explore sensor performance and 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. Simply monitoring its temperature/humidity/pressure exposure, its air-flow, and gas sensors of cross-sensitive pollutants could provide incredible 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 spuriously instead of predictably are inferior in design and construction.

For any of this to work, we need to compare our cheap sensor against a high quality reference, so we can learn when a sensor is in error, and what conditions may be indicitive 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. This is the first predictive 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 premise that its data *won't be reliable*, that we need to predict when it is. 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 reliabilty of cheap sensors, pushing the state-of-the-art forward. Validating this data would opening up a world of reliable, distributed data to the research community. In the process of testing this approach, we are building scalable solutions for data sharing and network interaction between cheap and expensive sensors. We are also engaging the communitiy with a new perspective on how to approach affordable sensing.

3. Related Work

Air Quality

Safecast

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 seen a lot of success and clout with their radiation data, and have just finished an alpha version of a stationary air quality monitor.

Aclima/Google

Google and Aclima have started a mobile, car mounted air quality project. However, their algorithms and hardware have been kept under wraps, and they likely intend to control the dataset. The speculation is that they are using a quite expensive setup in the back of the car, slowly drawing in air over time. Aclima has a reputation for litigious behavior, and it is generally understood that accurate mobile particulate sensing with a car-mounted system is still elusive.

Copenhagen Wheel Project

Out of the MIT SENSEable lab, it is mounted on a bike wheel and can help assist your pedaling, lock your bike, and capture data about air quality. The original project was unveiled in 2009, but it has since been licensed to a Cambridge Area startup (Superpedestrian) and is 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. The openSense team has published many excellent articles that address some of the fundamental problems in mobile air quality sensing and prediction with low cost sensors. They've laid the groundwork for dealing with low cost 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 viabile. However, some of their assumptions with this device included (1) ozone has limited and predictable spatiotemporal variation (which is true, much less than other pollutants), and (2) their preliminary tests on the effects of airflow revealed significant errors for high pollution levels.

Cheap Sensor Development and Testing Groups

There are organizations (such as SCAQMD 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). These groups are essential for verifying the claims of new products, and have great insight into the trends of success and failure with new sensor technology.

Other

Other projects worth mentioning are MIT's stationary/expensive clarity sensing project, the independent and nicely designed tricorder sensor platform, the innovative propeller health project that extrapolates pollution data from asthmatic inhaler usage, the open source and mobile UPOD project, the mobile Italian uSense project, the 'wearable' AirBeam kickstarter project, the Boston based startup Elm that translates their distributed air quality data into useable advice (and has an open invitation on their site to hook your sensor into their network), the Israeli BreezoMeter startup that extrapolates street-level/high resolution air quality data (presumably from wind patterns and known reference data), the London-based company cleanSpace that provides air quality maps and incentivizes air quality friendly commuting, the startups Clarity and Tzoa that are selling versions of the 'world's smallest' PM2.5 wearable, and the startup uHoo which is building a home air quality monitor. Other slightly less interesting, but still notable, include a handful of other hardware/commercial projects (the Air Quality Egg, CMU's Speck, Atmotube, AirboxLab, Airmon, SmartCitizen) and independent research like the French citizenAir project and DIY Public Labs work. 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 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, which is a database backend 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 necessarily 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 seperate 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 heirarchy 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. 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 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).

HAL, or Hypermedia Application Language, is another hypermedia semantic web data model with XML and JSON formats. 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. 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 is a standard for data sharing based on HAL and JSON. 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 make a large, coherent ecosystem realistic.

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. Several nice data visualization scripts have also been written on top of ChainAPI.

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 sacraficing functionality.

4. Overview of Design and Contributions

The are three main goals with the learnAir project. The first of 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 cheaper 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 low-accuracy citizen projects– and allows seamless, easy interaction between them. It also must support scalable, automatic support for the machine learning algorithms validated as part of our first goal. 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, provocative 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 a more accurate picture of personal exposure compared to comparably priced and speced 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, new predictions will improve and older predictions will be revised.

Machine Learning Validation

Sensor technology– especially in the air quality space– is getting cheaper and more reliable. For the first time, well-reputed sensors are starting to appear in the $100 range. However, there are limitations on what is possible with small, cheap 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 suceptibility 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 one, very clear 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 there 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 implement trustworthy, strong designs, but simply cannot afford to build a controlled environment around the measurement sample 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 our measurement.

To put this theory to the test, we built a stationary device that can measure ambient conditions (temperature, light, humidity, wind) with 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, right next to the inlet for expensive, reference EPA measurement equipment. The data collected from our measurements was used to compare against the 'ground truth'

provided by the EPA, 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), we can characterize how well our algorithms predict our sensors 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 instance 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 seperation of concerns, so that raw data and data processing scripts are seperated, instead of pre-processing occuring 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 anomolies, 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 a powerful 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 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, to 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, and 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 certainy 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 soley 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 Andriod applications. It has plugins to access the phone GPS, and a simple D3.js library was used for plotting. ChainAPI- our backend solution- supports websocket connections and a subscription model to push data to the cloud and recieve the lastest 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.

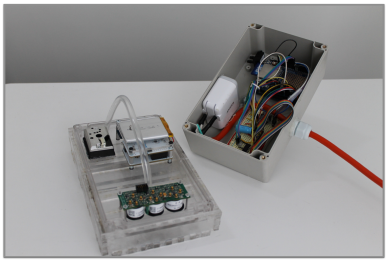


Figure 3: LearnAir Sensor installed at MassDEP site, opened.

The first sub-system is an off-the-shelf air quality monitoring system called the SmartCitizen Kit (frequently abbreviated SCK). 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 inclues 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,

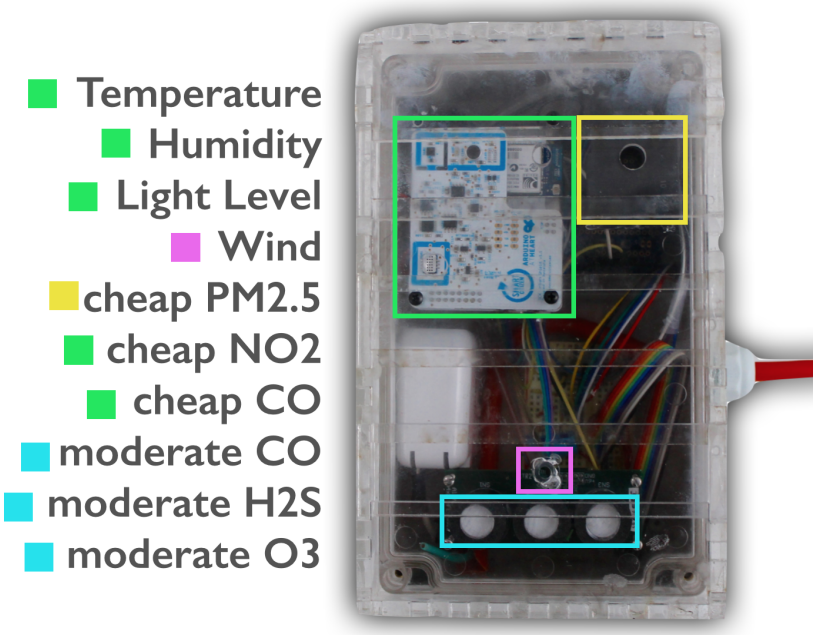


Figure 4: LearnAir Sensor installed at MassDEP site.

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.

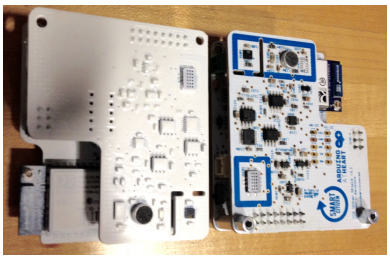


Figure 5: Corroded SmartCitizen Kit on the right, Conformal-coated new kit on the left. Relevant sensors on the new kit were taped off before coating to prevent contamination.

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 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. 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 Auxilary 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 (NO2 sensor picks up 30-60% of O3 at 100ppm). Other cross-sensitivites 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 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 incredible generosity and helpfulness of the Massachussets Department of Environmental Protection, we were given 24 hour 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 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 for quality assurance and re-calibrated every few weeks. 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 fall 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 in $\mu g/m^3$.

Wind speed and direction data was captured on a minute time-scale with a high-quality, vane style Met One 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.

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 0.5% precision, 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 O3 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 a stream the air with an O3 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 7. Our sensor and the inlet both face downward, however the Federal reference has active airflow. These inlets are mounted approximately 12 feet in the air, recessed from the street 30-40 feet.

Figure 7: LearnAir Sensor (box on left) installation next to MassDEP inlet (top of pole on right).

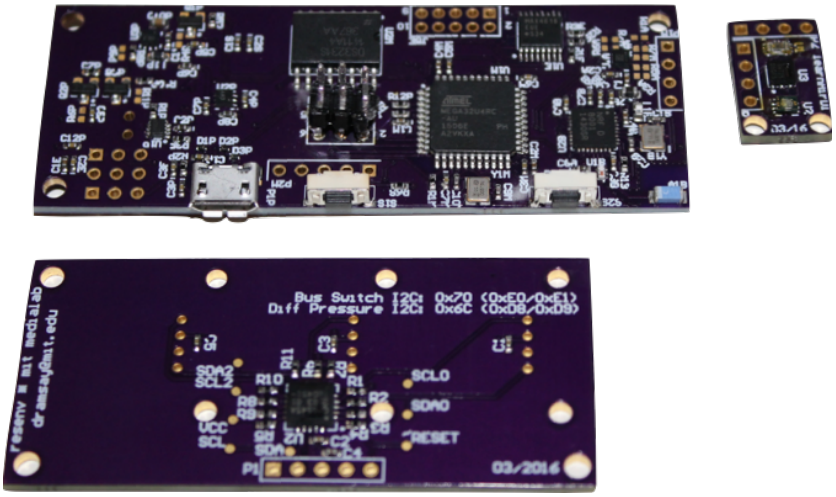


Figure 8: Main and daughter boards of learnAir V2.0.

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 8. 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 quiesent 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 seperates concerns when testing/verifying the circuitry.

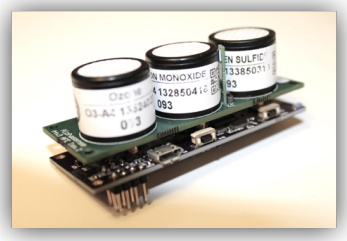


Figure 9: Second revision, Atmel based learnAir main board mated with the AlphaSense sensor frontend.

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 nicer $500 AlphaSense **OPC-N2**. The OPC-N2 takes advantage of mie scattering algorithms and uses nice optics, but it is insensitive to particules 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 does include 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 10). The

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

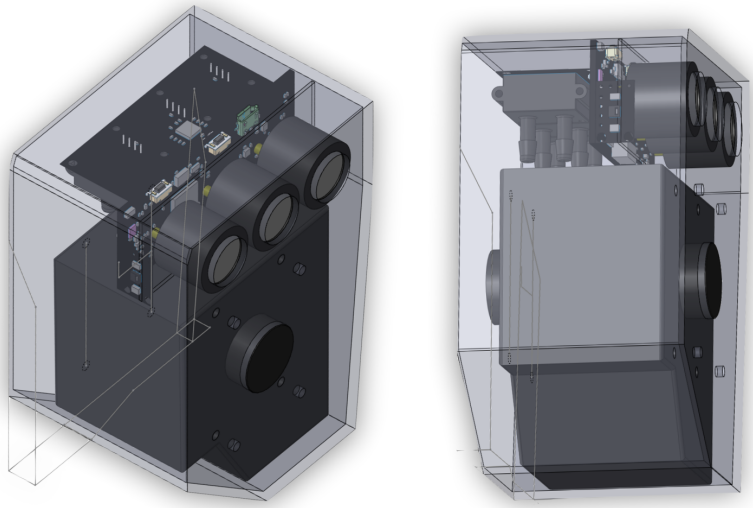


Figure 10: Final design of the portable system.

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.

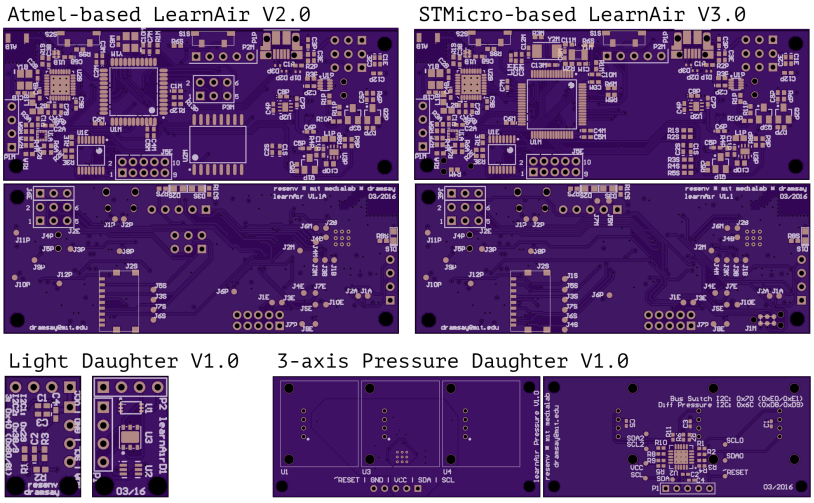


Figure 11: Layouts for revisions 2.0 and 3.0 of the learnAir board.

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. This type of advanced power optimization is a goal of learnAir V3.

This 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.

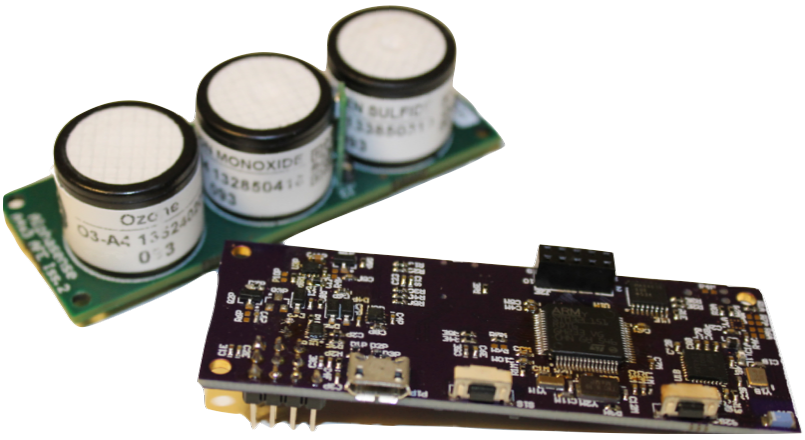


Figure 12: Third revision, STM32L152-based learnAir main board next to the AlphaSense sensors.

Hardware Comparison and Analysis

MassDEP hardware uses robust, FEM certified techniques– thus it can be taken as 'ground truth' over the time-scales it measures. The corresponding air quality sensors in the learnAir system are variable in quality, and their analysis resides 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 the sensor readings reported from the learnAir device against corresponding weather API data retrieved from ForecastIO for the latitude and longitude of the device. Only temperature and humidity fall into this category (both available from the device as well as from an independent API). 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 14 shows a comparison of each SmartCitizen reading with each ForecastIO value– ideally they would be always be identical, falling on the diagonal dotted line. 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 agreement between the measurements is quite good, and suggests trustworthiness.

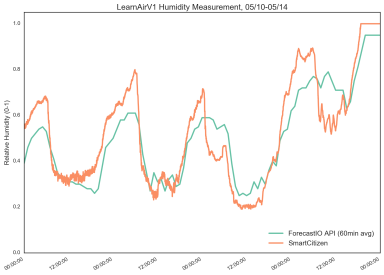


Figure 13: Humidity Comparison of SmartCitizen (orange) and ForecastIO (green) over 4 days

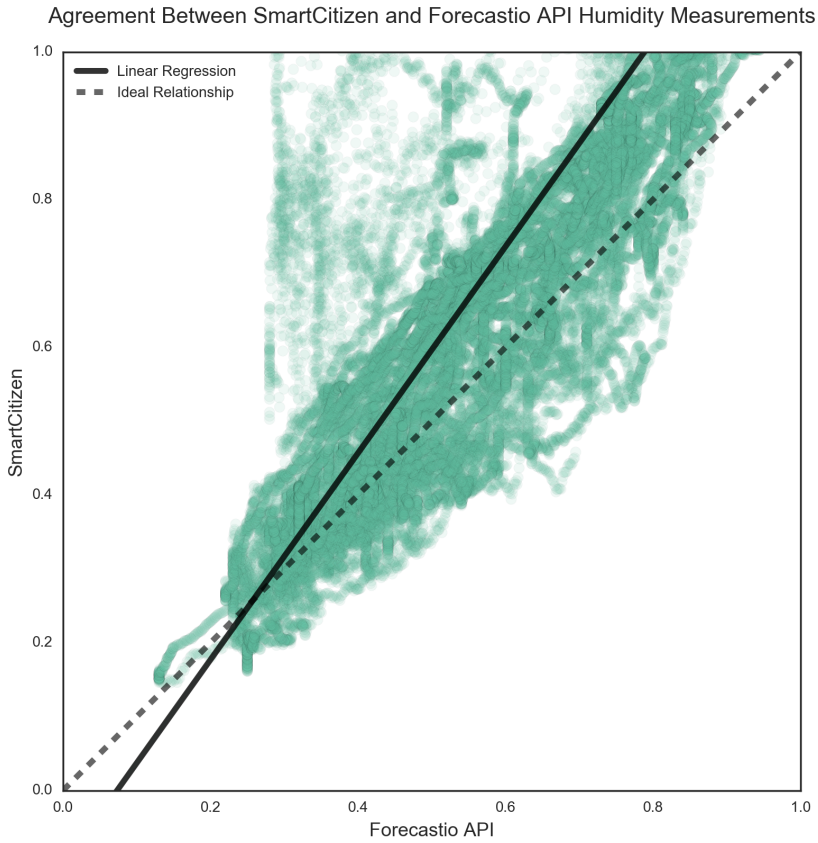


Figure 14: Humidity Comparison, SmartCitizen and ForecastIO

Temperature was recorded inside the box by both the AlphaSense temperature sensor and the SmartCitizen Kit. Figure 15 shows quantization error on the raw AlphaSense reading (teal)– this is mitigated by taking a 15-minute rolling average (orange). In Figure 16 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 Celcius. 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.

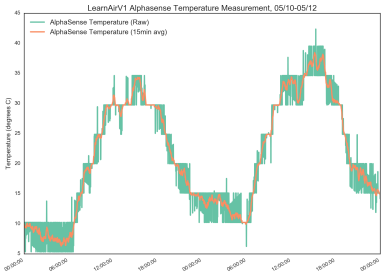


Figure 15: Alphasense Raw Temperature Data (green) with 15-minute averaging (orange)

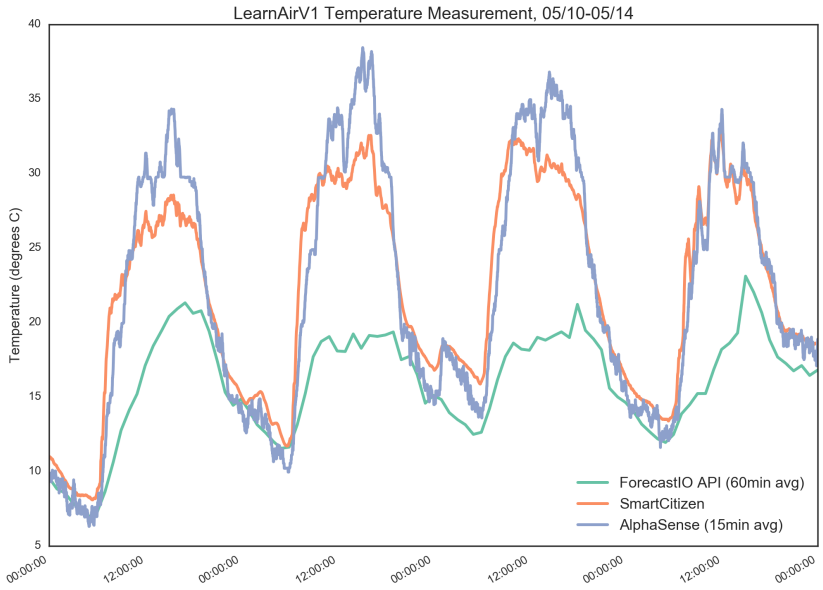


Figure 16: Temperature Close up

Wind

The differential pressure sensor was a small, cheap, and experimental way to measure airflow and wind through the device. While the learnAir V1 sensor we built 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 meaurement. 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.

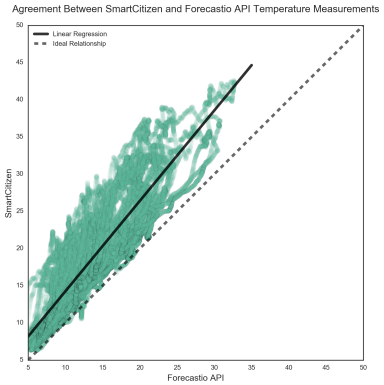


Figure 17: Temperature Comparison, SmartCitizen and ForecastIO

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 ex-

pect air coming directly at the sensing face of the device to penetrate much more than air flow coming at the back.

Figure 18: A picture of a simple laminar flow test setup for rough wind directivity characterization.

To test this, PVC filled with large straws was placed in front of a fan to approximate laminar flow conditions (Figure 18). Cardboard was placed around the laminar flow section to prevent spurious eddie 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. Figure 19 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). Figure 20 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 artifically affect how sensitive to directional sources of pollutions 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.

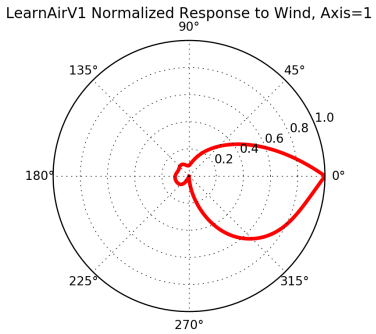


Figure 19: Wind Directivity Polar Pattern, Axis = 1

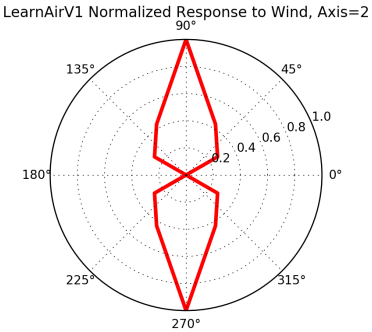


Figure 20: Wind Directivity Polar Pattern, Axis = 2

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 gainst the MassDEP data for one day (after pre-

conditioning the signal and LMSE scaling it against the MassDEP reference). There is clear correlation in overall trend, suggesting the pressure sensor is capturing meaningful information. Tight agreement of ±5% between readings is highlighted in green.

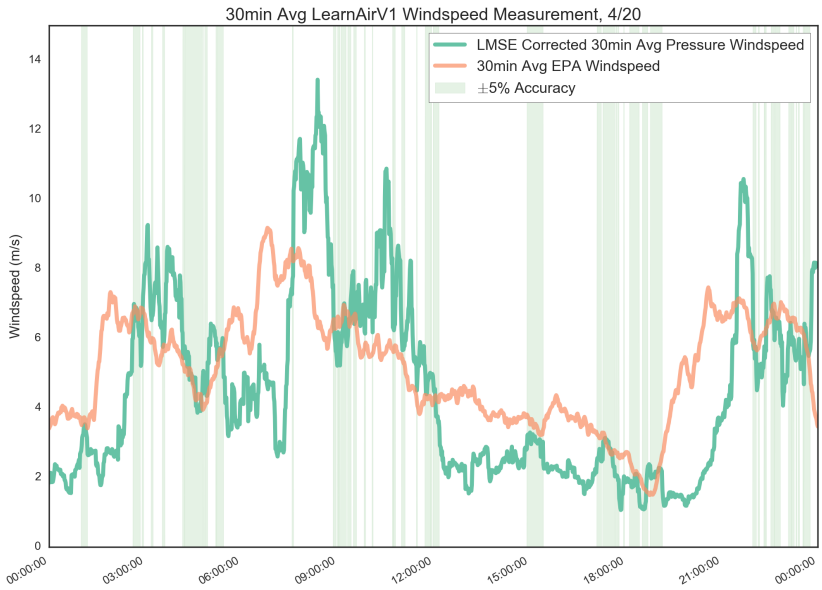


Figure 21: Wind Speed Measurement with 10% Accuracy, Zoomed

While the overall trend is there, it is clear that there are some large discrepencies. Based on our polar plots, it seems worthwhile to look at the error 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 error and wind direction– there are large peaks in error when the wind approaches from 60, 120, 210, 270 degrees, while 0, 90, 180, 260, and 320 degrees seem to have low error rates. This does not match with expection from our polar plots exactly (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), and hints at 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. There is a wealth of future research work necessary to optimize sensor geometry, orthogonalize sensor

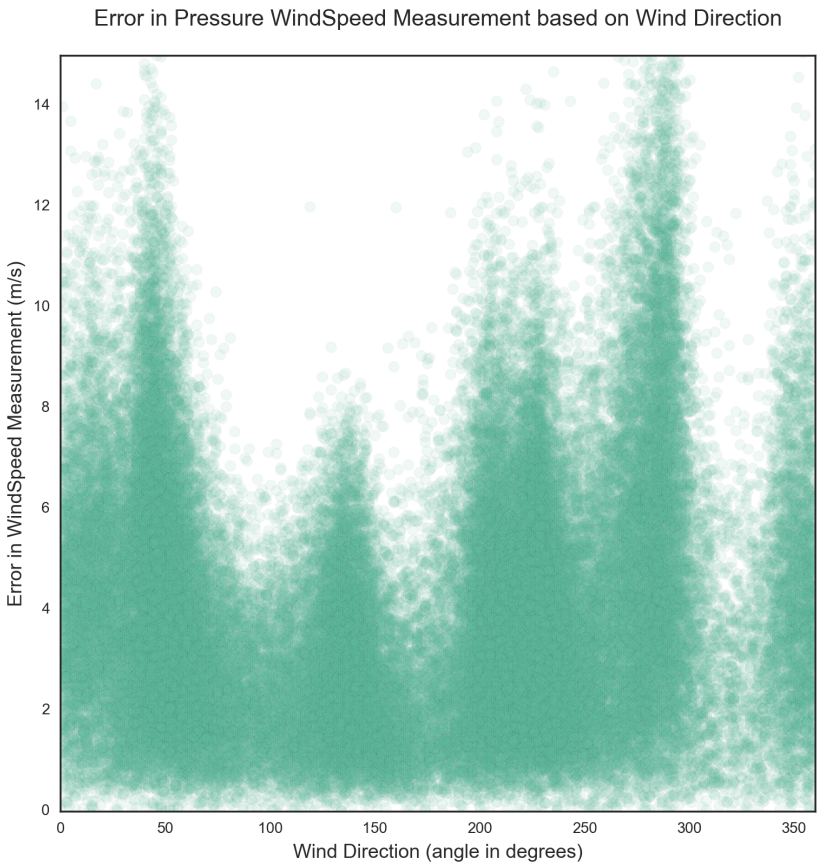


Figure 22: Windspeed Error vs Wind Direction

axes, characterize the effect of device geometry and turbulence on system linearity, and further understand the wind sensing potential for this technology. For this project, the pressure signal appears to provide insightful, near-field air flow information that we used as a training feature our machine learning model. More figures describing the wind data can be found in the Appendix.

6. *ChainAPI for Air Quality*

ChainAPI 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.

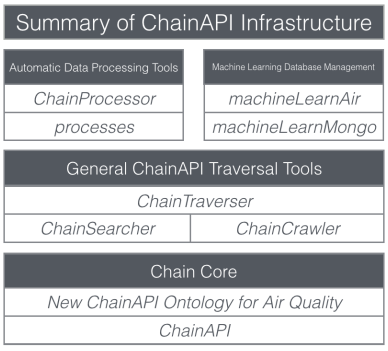


Figure 23: Summary of New ChainAPI Infrastructure.

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. 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 installation. 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 structure 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 an 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 meant 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.

After defining a new ontology for ChainAPI and creating a development environment that takes advantage of this ontology, several tools to enable scalable interaction and advanced learning were created. 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 data processing, as well as scalable, advanced machine learning techniques. It encourages programmatic data processing that can scale and is easily trackable. It encourages a separation of concerns– so the best-in-class raw data collection and the best-in-class pre-processing 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 acccess 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 minutae are important for interoperability. Timestamps are stored using ISO8601 standards (timezone aware UTC). 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 maintanence.

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 indiciation 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. 'Celcius', '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.

```
crawler = ChainCrawler('http://learnair.media.mit.edu:8000/',
    ↪ found_set_persistence=2, crawl_delay=500)

#--- Blocking Find ---#
    x = crawler.find(namespace='http://learnair.media.mit.edu:8000/rels/',
        ↪  resource_type='sensor', resource_extra={'sensor_type':'
        ↪ Alphasense03-A4'})
    print x

#--- Threaded Queue ---#
testQueue = Queue.Queue()
crawler.crawl_thread(q=testQueue, namespace='http://learnair.media.mit.edu
    ↪ :8000/rels/', resource_type='Device', resource_title='test004')

#caution: this main loop doesn't end
while True:
    uri = testQueue.get()
    print uri

time.sleep(5)

#--- ZMQ Socket Push on TCP://127.0.0.1:5557 ---#
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 chain, and then intelligently traverse local relationships within the Chain 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)

Exhuastive 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.

```
searcher = chainSearch.ChainSearch('http://learnair.media.mit.edu:8000/
    ↪ devices/10')

#-- Find Creation Link to Make a Sensor on this Device --#
resource_uri = searcher.find_create_link(namespace='http://learnair.media.
    ↪ mit.edu:8000/rels/', resource_type='sensor' )
print resource_uri

#-- Change Starting Point for Search --#
searcher.reset_entrypoint('http://learnair.media.mit.edu:8000/devices/?
    ↪ site_id=1')

#-- Find First Device with Name 'Device #3' within 4 Link Steps --#
resource_uri = searcher.find_first(resource_title='Device #3', degrees=4)
print resource_uri

#-- Find All Devices within 2 Link Steps --#
list_of_resource_uris = searcher.find_degrees_all(resource_type='Device',
    ↪ degrees=2)
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 cascading child resources along a pre-defined link path. It can also pull and push data safely 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 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 formated 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 deploymet
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 provide a prompt to the user in which they can simply 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':'O3_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 class 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/pr (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 querys the process file to see if additional local data is required for the data processing step, and then mediates datafow 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
```

```
8  from .. import machineLearnDatastore
9
10 def dispatcher(metric, unit):
11     #this tells which extra data are required and which functions to use
12     #to process a given metric/unit combination for this sensor type
13
14     return {
15         'temperature_raw': { 'raw': {
16             'function': raw_to_temp
17             }},
18         'temperature': { 'celcius':{
19             'extra_data': ['humidity_corrected', 'light_corrected'],
20             'function': temp_to_learned_temp
21             }}
22
23             }.get(metric, None)[unit]
24
25
26 #all functions should return [sensor_type, metric, unit, data_to_post]
27
28 def raw_to_temp(data):
29     #processes raw SHT21 readings to accurate temperature using datasheet
30     #equation posts to a 'SHT21Temperature' sensor measuring 'temperature'
31     #in 'celcius' units on the same device as the 'raw' SHT21 temp. data
32
33     for data_element in data:
34         data_element['value'] =  -50.0 + 175.72 * (data_element['value'] *
               ↪  10 / (2**(16) ) )
35
36     return('SHT21Temperature', 'temperature', 'celcius', data)
37
38
39 def temp_to_learned_temp(data):
40     #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', 'celcius_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 auxilary 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 auxilary 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.

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 the machineLearnMongo is indexed by timestamp, latitude, and longitude.

The most important part of machineLearnMongo are 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 pull 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.

---

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 unspecificed (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 unspecificed (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 in memory, 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 collection, 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.

```
1 from machineLearnDatastore import *
2
3 learner = machineLearnAir(collection_name="AlphaSenseNO2" ,
    ↪ update_model_with_x_new_entries=500)
4
5 post_data = learner.run(input_data, algorithm='svm')
```

```
1 machineLearnAir.py
2
```

```
<...>

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 allows a scalable, distributed ecosystem, that supports easy interoperability while keeping the barriers to entry low.

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 new infrastructure. We wrote several new tools 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 seperation 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 reliabilty based on all of the latest ChainAPI data.

7. Data Analysis and Machine Learning

In the last two chapters we outlined the portable hardware that was built to measure air quality and push it up to ChainAPI. We discussed our new backend 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.

For this test, we were able to co-locate six types of cheap 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 season to the summer).

This experimental design gives us the ability to take the difference between our cheap 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. 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.

<Summary of interesting take aways and results.>

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.

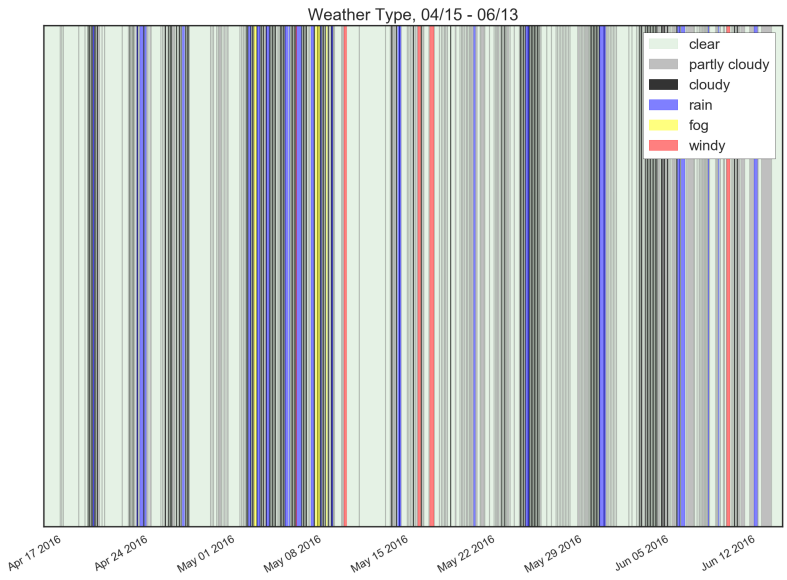


Figure 24: Weather during Test Period

From April 15th through May 23rd (38 days with one 40 minute service break), an older set of AlphaSense O3 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 O3+NO2, NO2, 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 (O3 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.

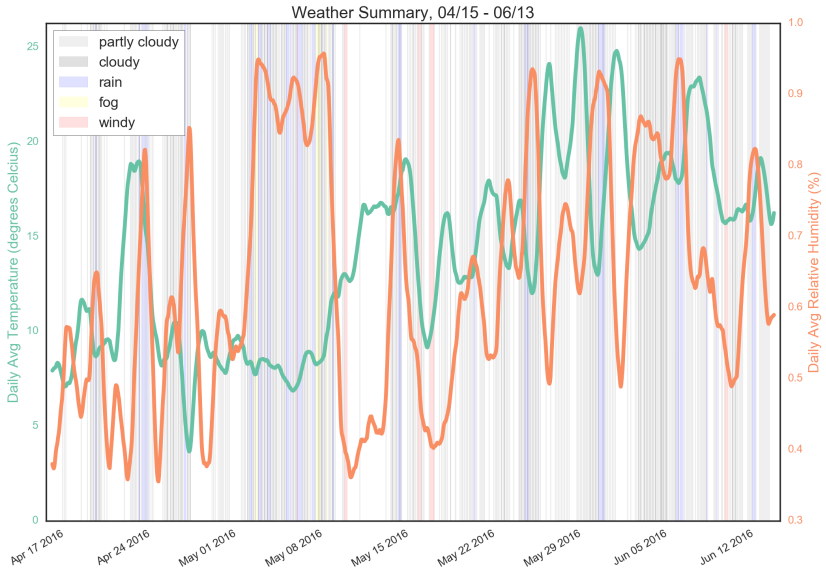


Figure 25: Temperature and Humidity during Test Period

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. 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 knowing the precise time the measurements started and ended). 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 minute 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 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 val-

ues. The formula for most of their sensors is:

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

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:

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

.

While we used the provided calibration data– as well as simple LMSE scaling– we found the best agreement came from LMSE minimization against the reference values 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$2-5% of the full range, though a larger tolerance was chosen for particulate), and each reading was 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 regression 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. The accepted standard for this 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. This will agree with our original confusion matrix if we set our probability threshold at 50%.

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.

Machine Learning Features

Overview of Features- API, measured, minute. Difference for Particulate hourly.

API calls, how we process

condition raw values from normal sensors too

In most of these applications, around 150 features were used for learning. Many of these were derived features from a core set of measured and observed features.

The following features table represents all of the features used. In general, some small subset of features were removed for each training session.

For instance, the higher quality Alphasense CO sensor was removed as a feature when training the SmartCitizen CO sensor. Training with this feature gives incredibly high accuracy at predicting failure, because it is effectively trained with the answer. By training with all similar quality, 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 feasible as a portable, cheap sensor, if this figures prominently in determining a sensor's failure, we can infer something about how the sensor is failing, and what type of sensor we might want to pair it with. In this way, the machine learning process is less evaluative of a current system's success and more exploratory and informative about a system's potential design.

The main sources for features were: sensors on the SmartCitizen board, the air quality sensors themselves (which gives an indication of cross-sensitivity, or a self-range where the sensor is untrustworthy), and data harvested from the Forecast.IO API.

Below are a list of the main features. Derived features from averaging over different time windows and taking derivatives were added to this list.

Features from Weka. Features plots.

Create differential measurements - temp in the box vs. out of the box, as well as humidity in the box vs. out of the box.

Create derivative features for most main features, in case rate of change causes issues for sensors (we know it does). We also looked at intelligently chosen averages over time- ones that helped minimize quantization errors, helped smooth 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.

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

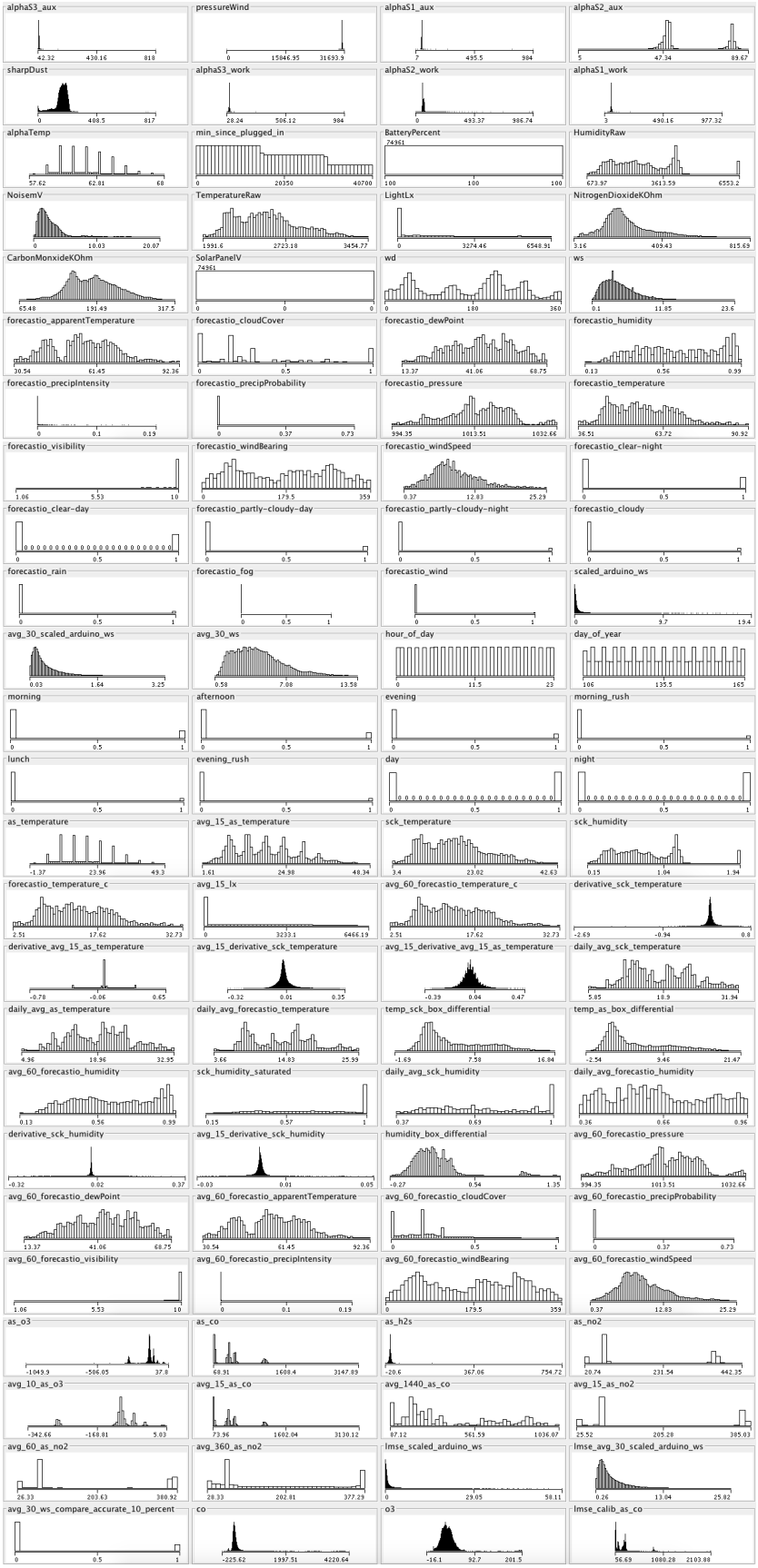


Figure 26: All ML features plotted with WEKA Tool

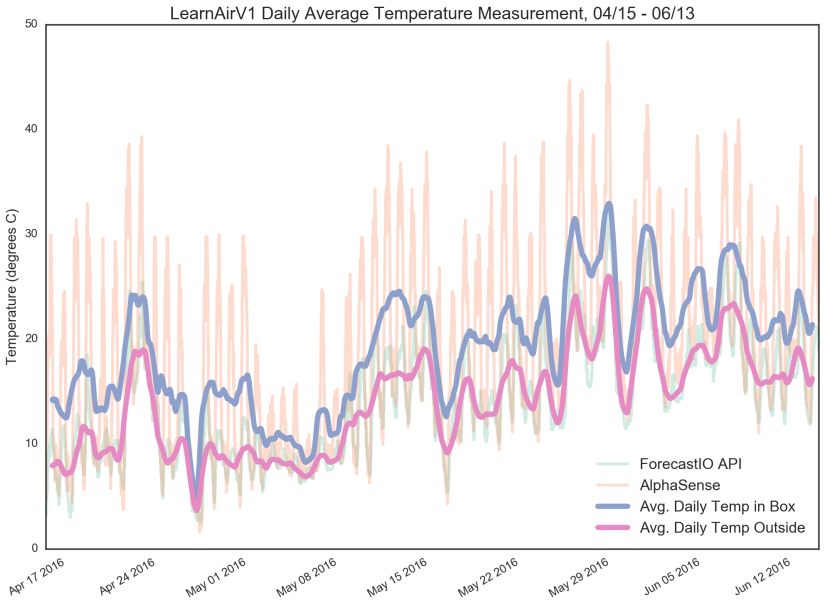


Figure 27: Temperature during Test Period

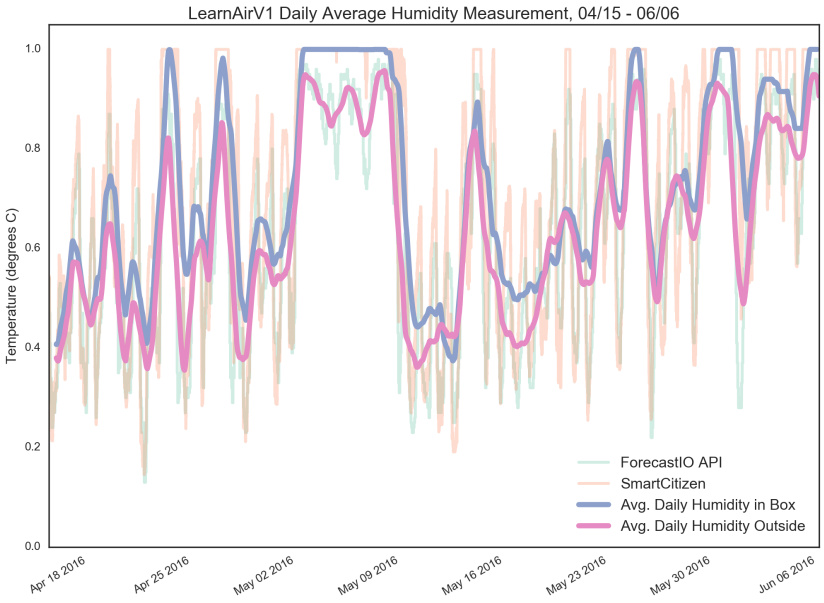


Figure 28: Humidity during Test Period

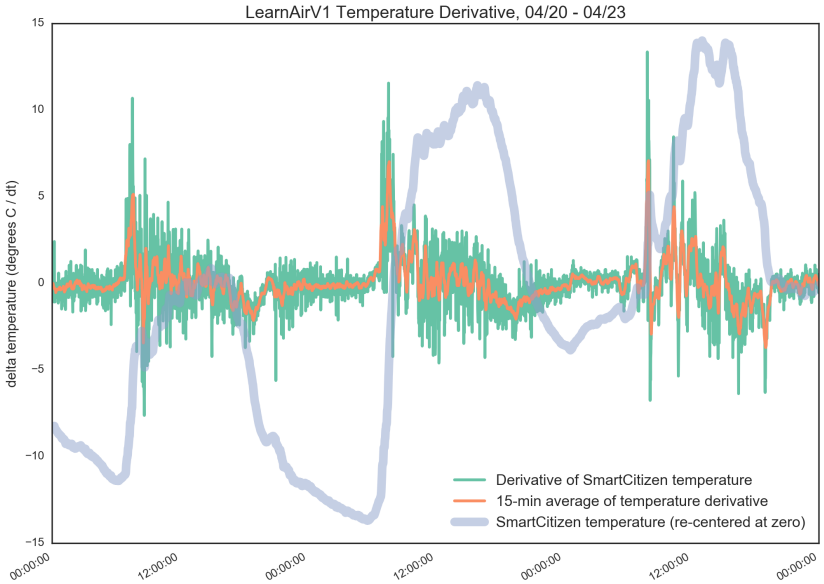


Figure 29: Temperature Derivative Feature Creation

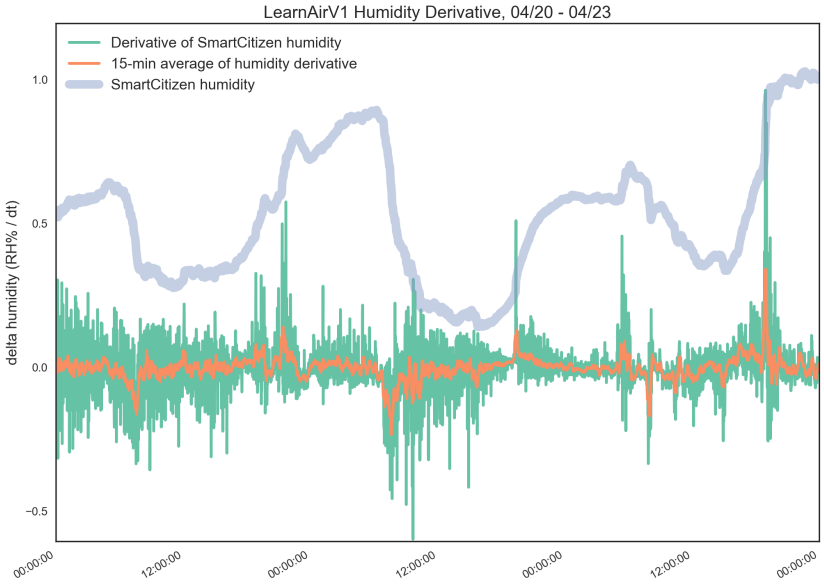


Figure 30: Humidity Derivative Feature Creation

SmartCitizen CO

This is about the smartcitizen CO results.

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.

Pre-processing

Talk about process for taking raw aux/working electrodes and making the basic calibration data.

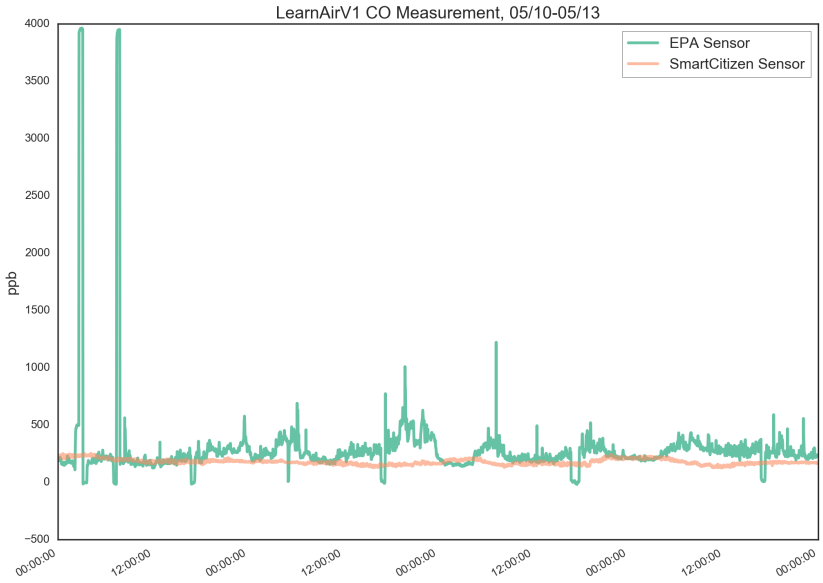


Figure 31: SmartCitizen CO Raw Data

Machine Learning

parameters = 'C':[0.001, 0.1, 10, 1000], 'penalty':('L1', 'L2') , 2 fold

===== best ROC_AUC score 0.81902438871

===== best params 'penalty': 'L1', 'C': 10

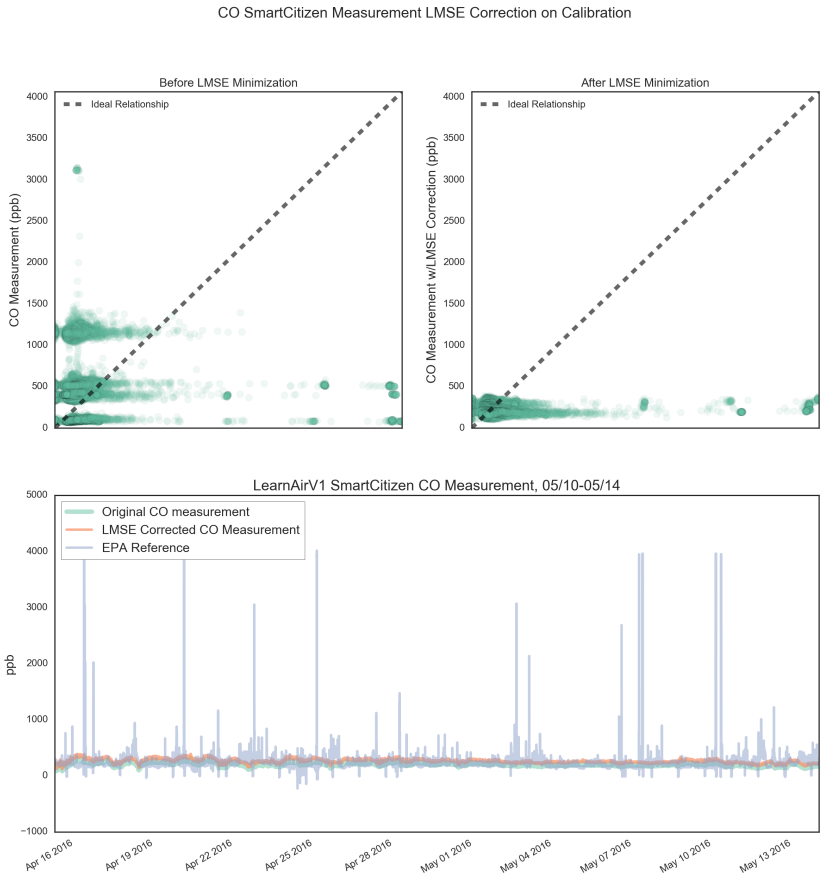


Figure 32: SmartCitizen CO after LMSE Calibration

here's text referencing the (Table 16).

here's text referencing the (Table 17).

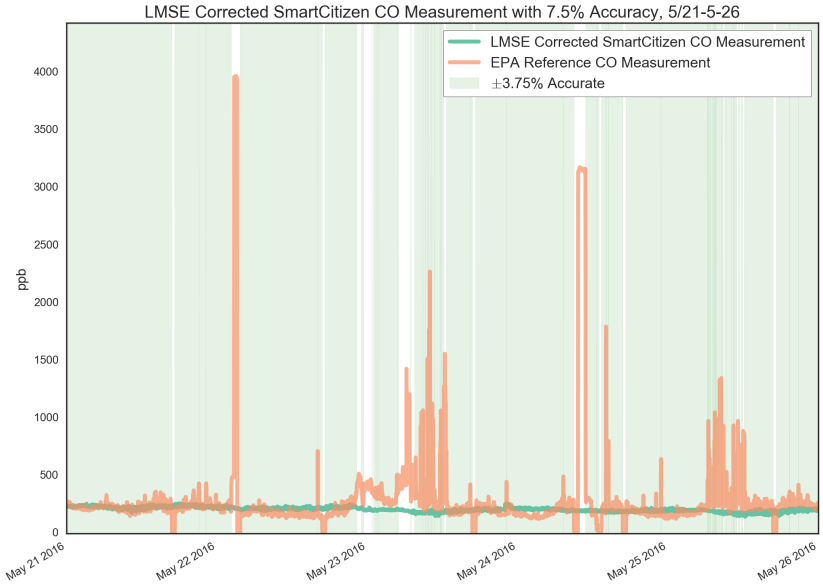


Figure 33: SmartCitizen CO with 7.5% Accuracy Threshold

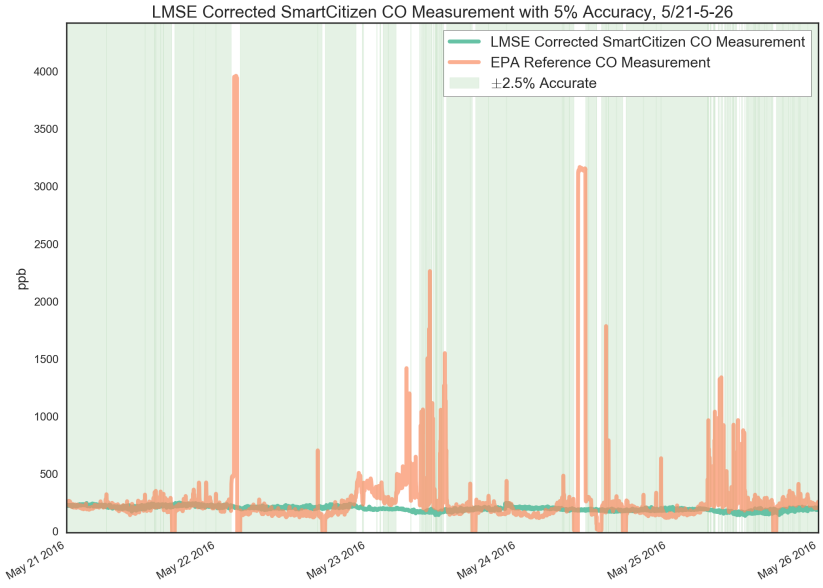


Figure 34: SmartCitizen CO with 5% Accuracy Threshold

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

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

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

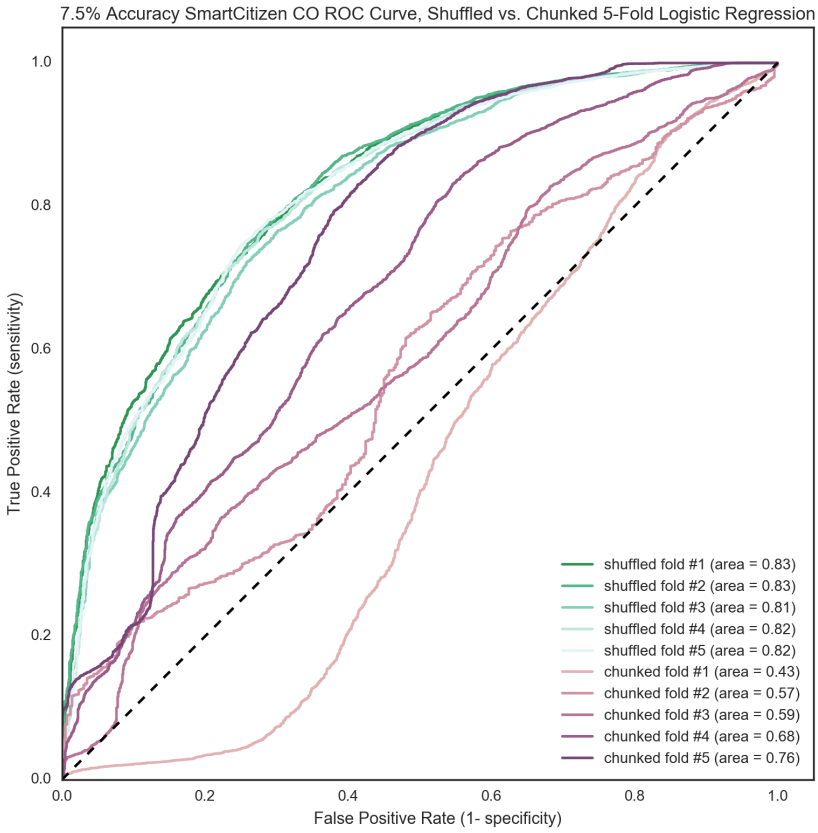


Figure 35: SmartCitizen CO ROC Curve

Table 4: 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

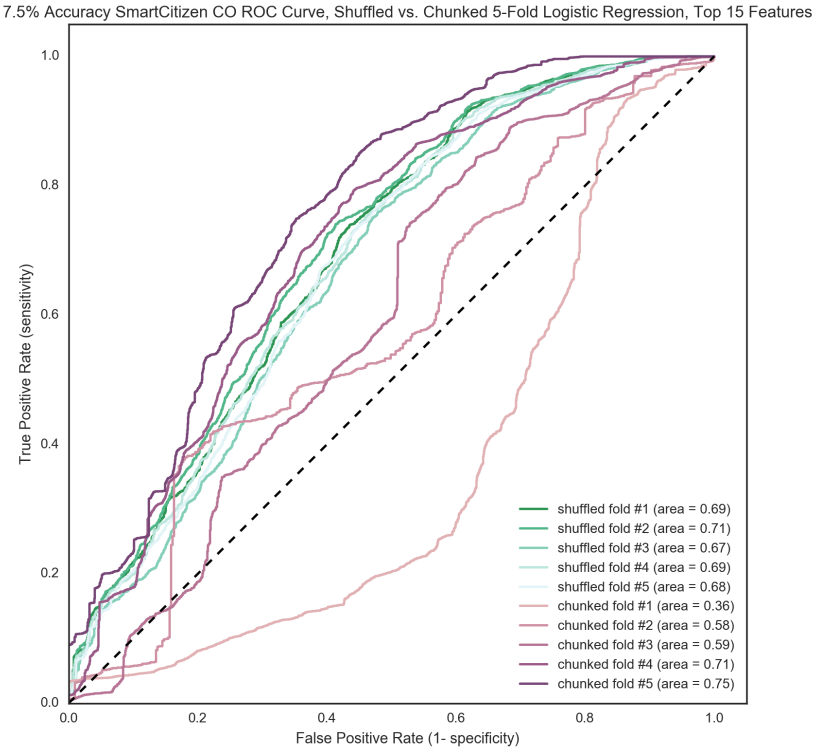


Figure 36: SmartCitizen CO ROC Using Top 15 Features

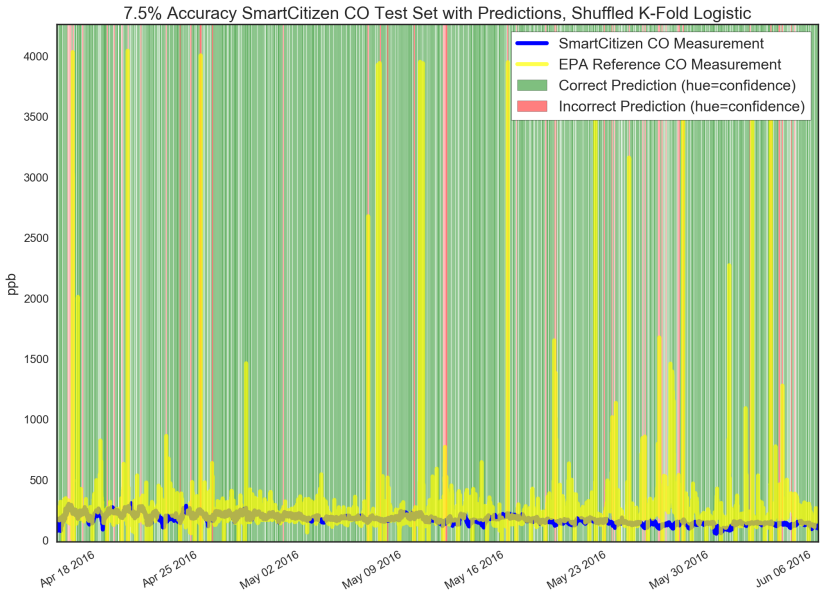


Figure 37: SmartCitizen CO Prediction Accuracy

Table 5: 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

SmartCitizen NO2

This is about the smartcitizen NO2 results.

One SmartCitizen NO2 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.

Pre-processing

Talk about process for taking raw aux/working electrodes and making the basic calibration data.

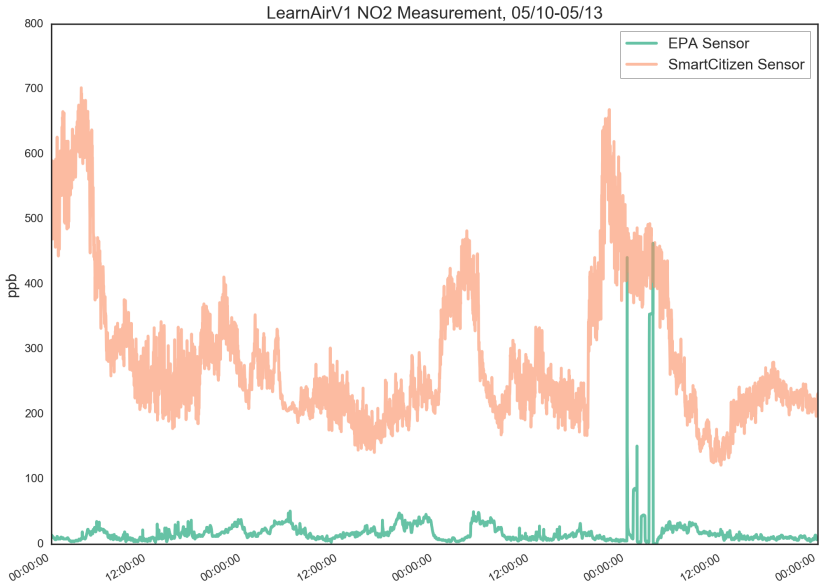


Figure 38: SmartCitizen NO2 Raw Data

Machine Learning

parameters = 'C':[0.001, 0.1, 10, 1000], 'penalty':('L1', 'L2')

===== best ROC_AUC score 0.907260610395

===== best params 'penalty': 'L1', 'C': 1000

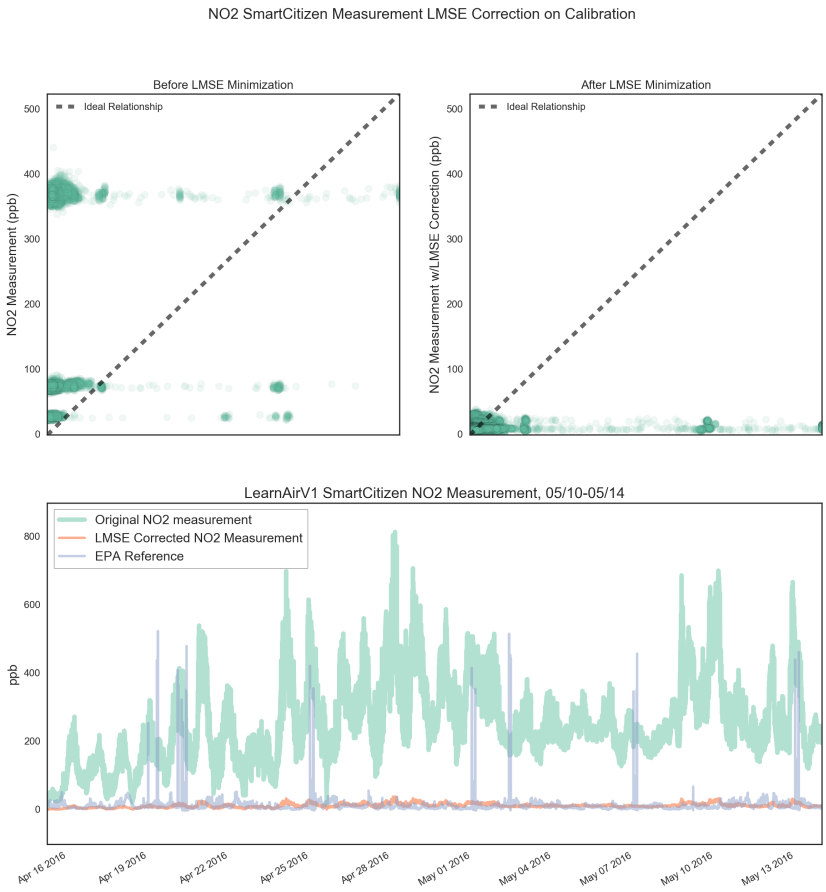


Figure 39: SmartCitizen NO2 after LMSE Calibration

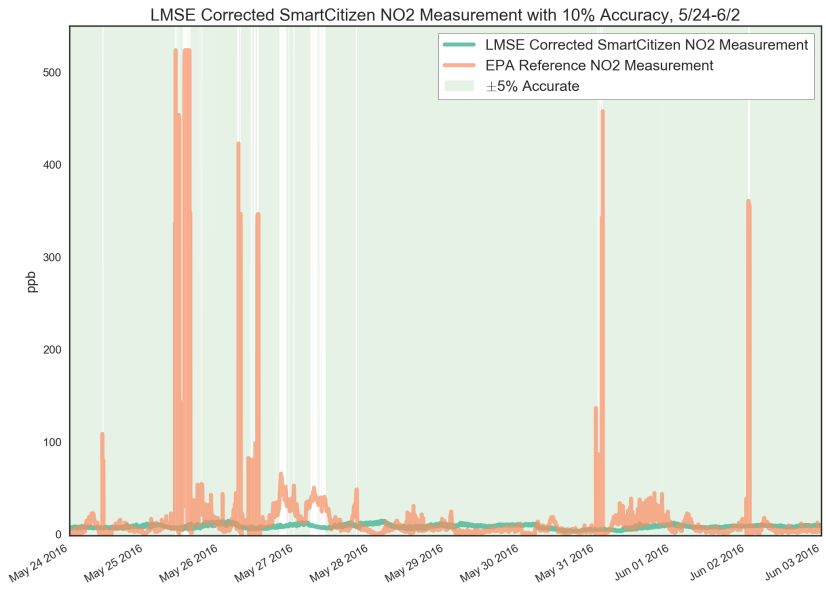


Figure 40: SmartCitizen NO2 with 10% Accuracy Threshold

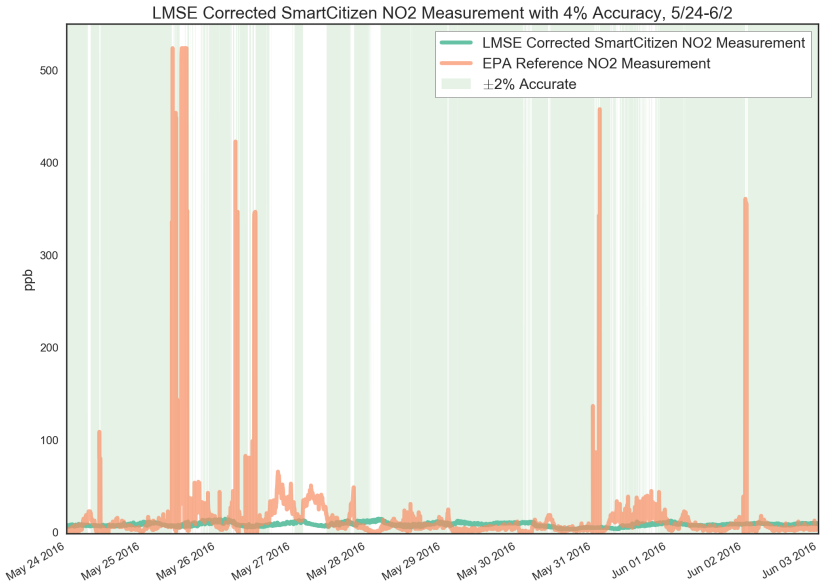
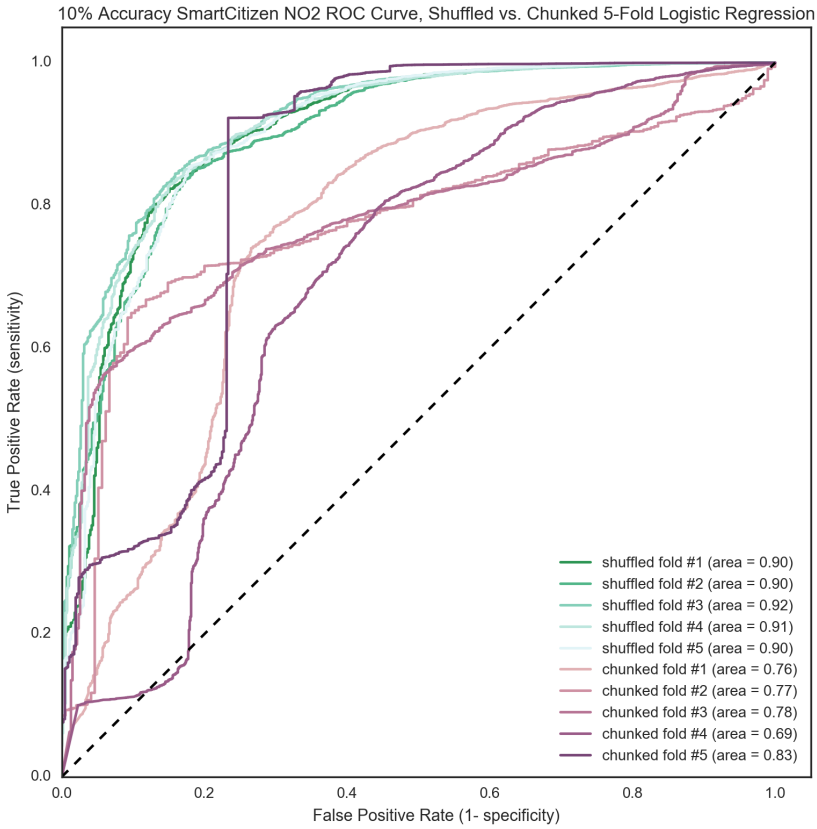


Figure 41: SmartCitizen NO2 with 4% Accuracy Threshold

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 6: Error Rates for Predicting SmartCitizen NO2 Accuracy with Logistic Regression



H

Figure 42: SmartCitizen NO2 ROC Curve

		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

here's text referencing the (Table 8).

here's text referencing the (Table **??**).

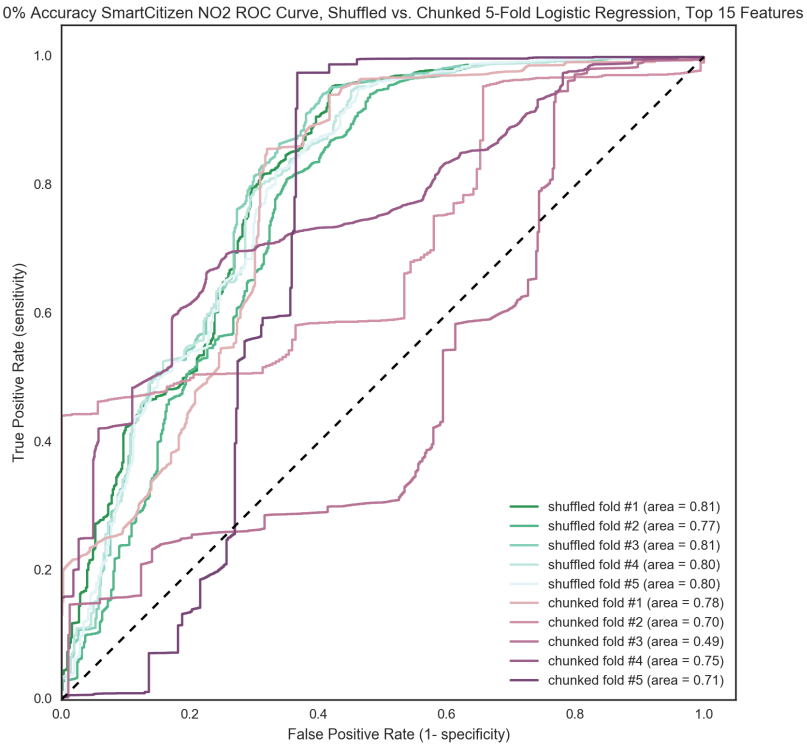


Figure 43: SmartCitizen NO2 ROC Using Top 15 Features

Table 8: 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

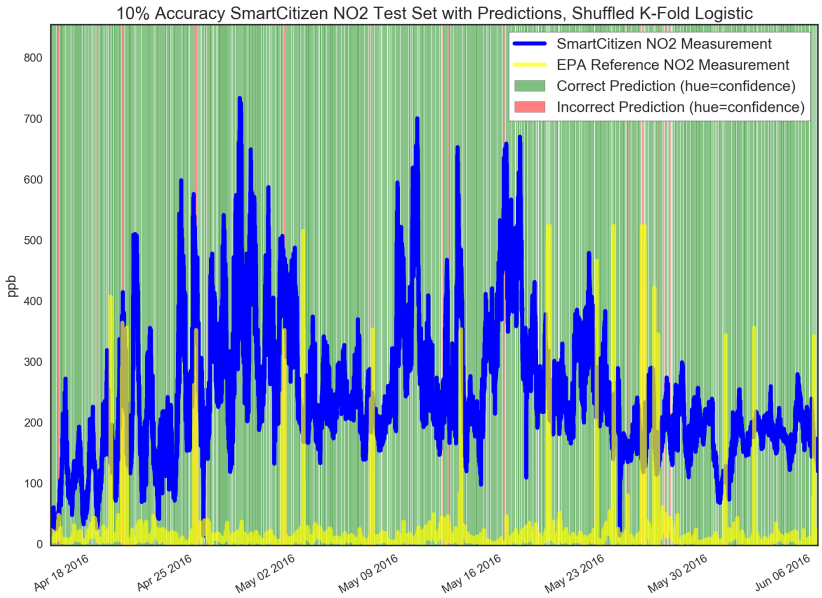


Figure 44: SmartCitizen NO_2 Prediction Accuracy

Table 9: 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

Sharp Dust Sensor

This is about the Sharp Particulate Sensor results.

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

Talk about process for taking raw aux/working electrodes and making the basic calibration data. 10am reading of EPA sensor is recorded on filter paper from 10a-10:50a, then measured with Beta Attenuation from 10:50-11a. Thus, we averaged the 50 minute readings starting on the hour and threw away the last ten minutes. Added some feature averaging on the 6hr , 12hr, day scale.

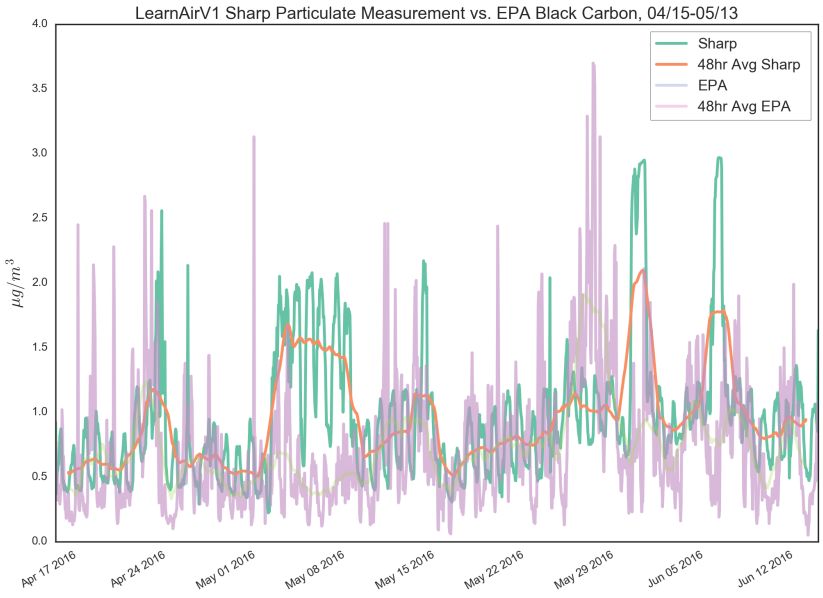


Figure 45: Sharp Raw Particulate Data

Machine Learning

parameters = 'C':[0.001, 0.01, 0.1, 1, 10, 100, 1000], 'penalty':('L1', 'L2') , true 5 fold validation

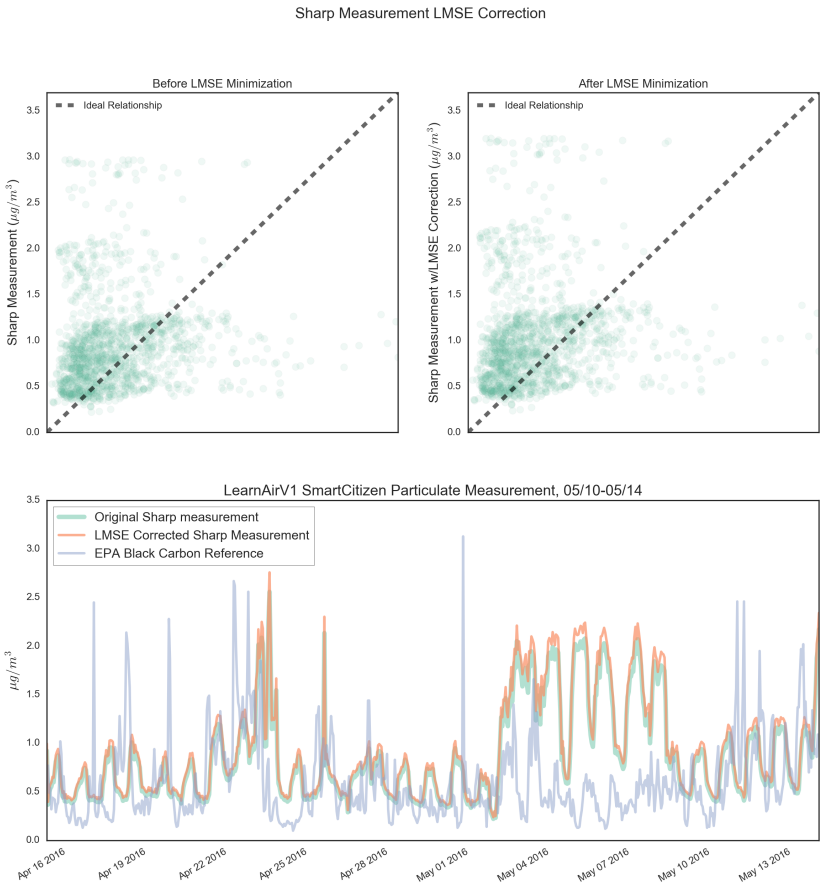


Figure 46: Sharp Particulate LMSE Calibration

===== best ROC_AUC score 0.868191734814

===== best params 'penalty': 'L1', 'C': 1

here's text referencing the (Table 12).

here's text referencing the (Table 13).

End Text for Sharp Section.

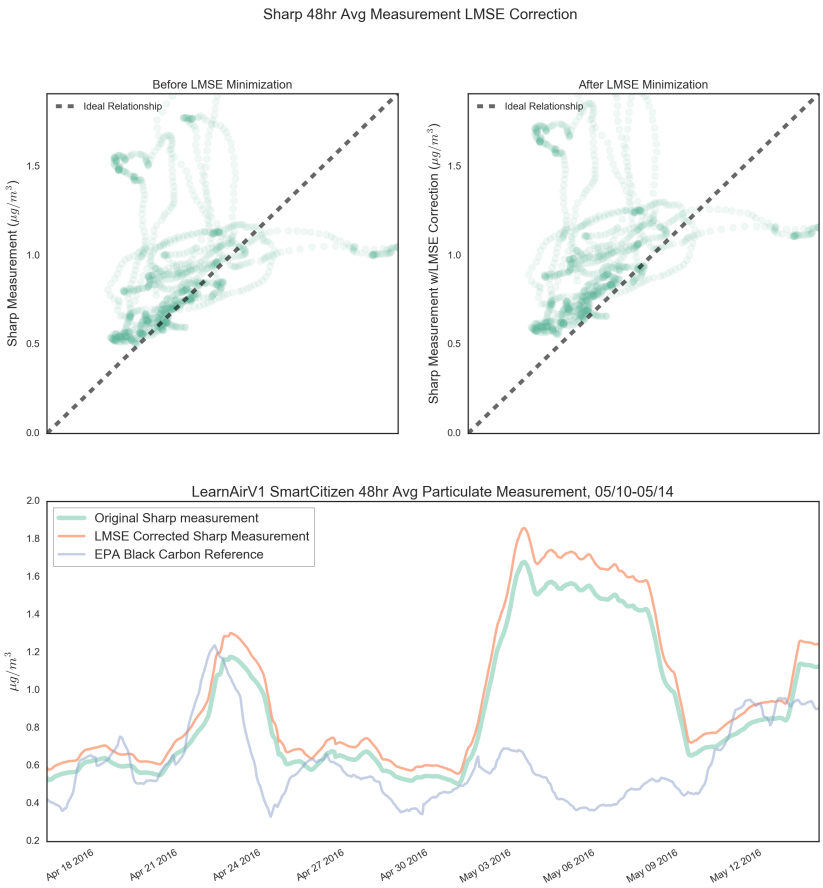


Figure 47: 2 day Average Sharp Particulate LMSE Calibration

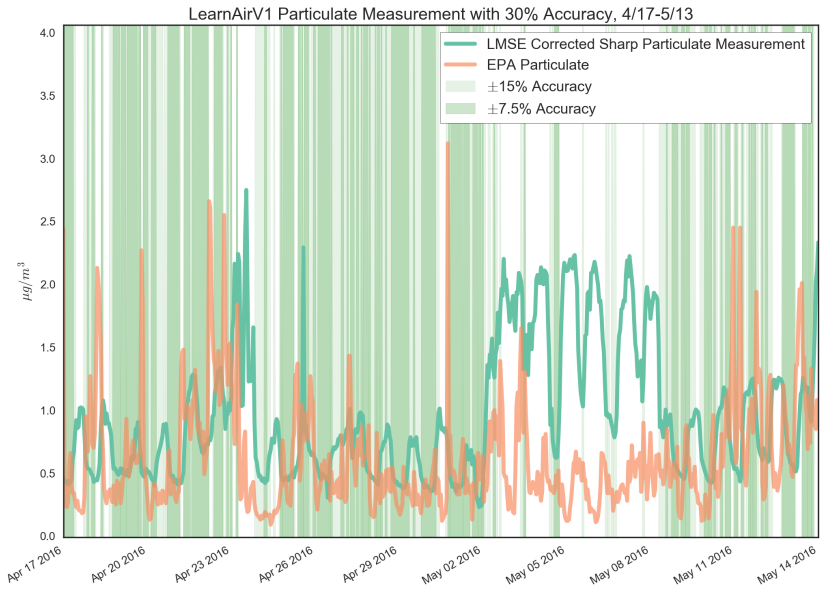


Figure 48: Sharp Particulate with 30% Accuracy Threshold

Error Rates for CO Sensor 1 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 10: Error Rates for Predicting Sharp Accuracy with Logistic Regression

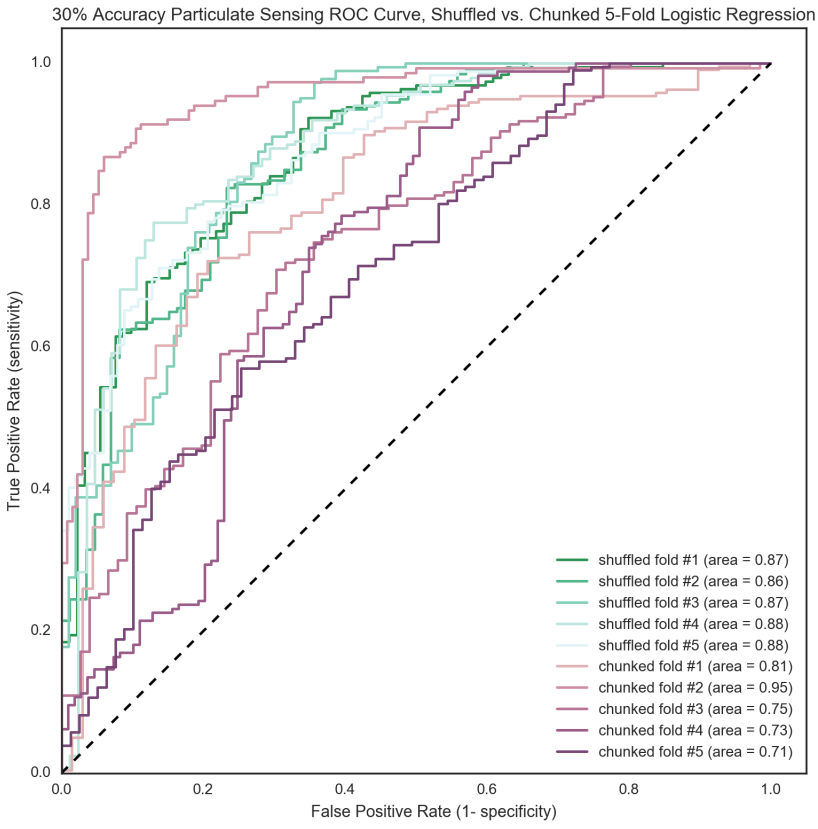


Figure 49: Sharp Particulate ROC Curve

Table 11: Average Sharp Particulate Confusion Matrix w/Shuffled K-Fold

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

Table 12: 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 (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

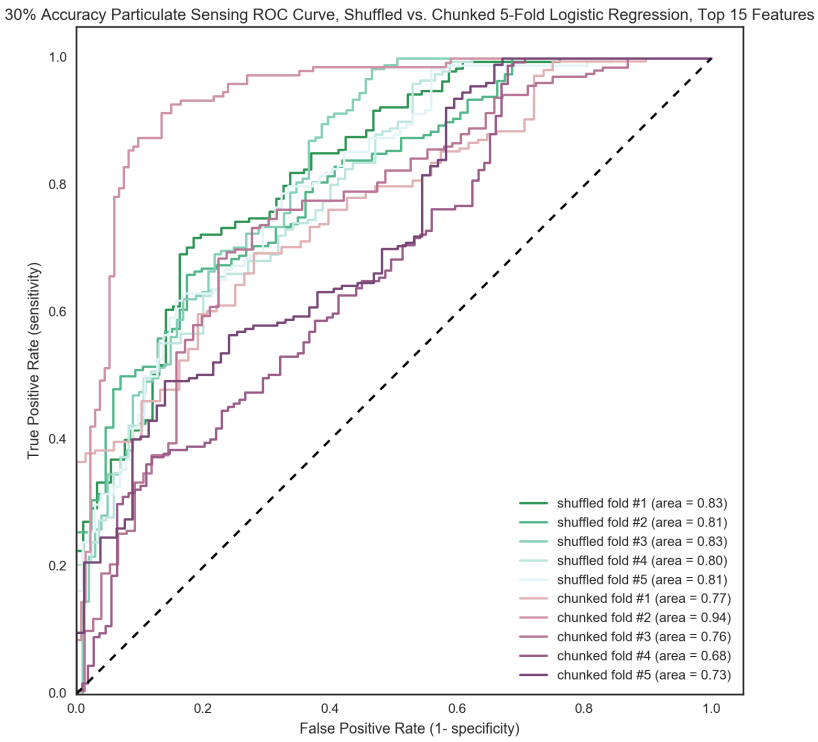


Figure 50: Sharp Particulate ROC Using Top 15 Features

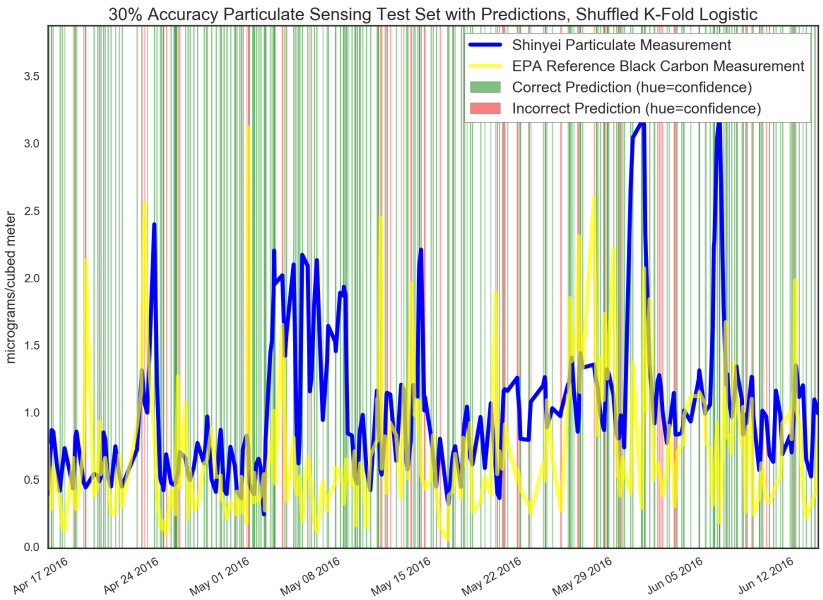


Figure 51: Sharp Particulate Prediction Accuracy

Table 13: 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

AlphaSense CO

This is about the alphasense CO results.

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.

Age is an important distinction between the two sensors, which makes this an interesting comparison. Additionally, while the first CO sensor was only calibrated for CO measurement.

Pre-processing

Talk about process for taking raw aux/working electrodes and making the basic calibration data.

Old sensor, raw data with simple calibration based on datasheet.

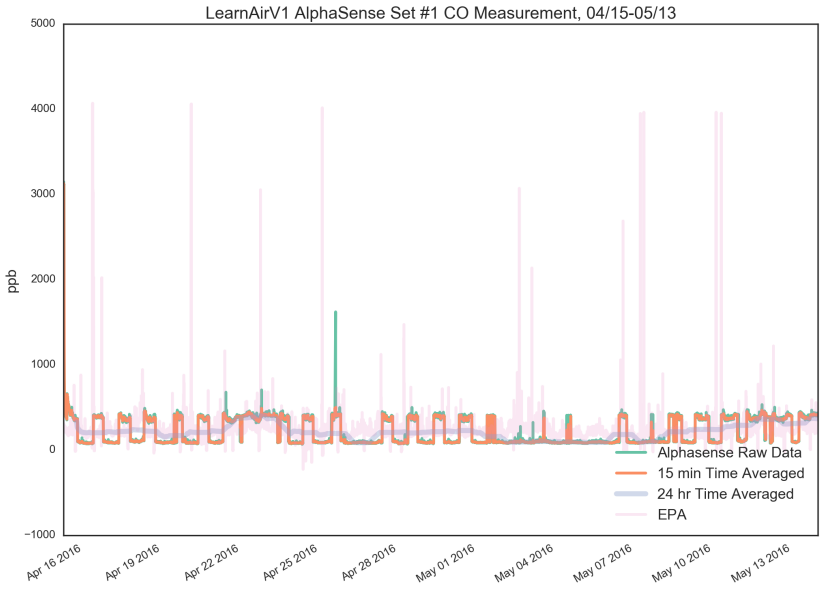


Figure 52: AlphaSense CO Sensor 1 Raw Data

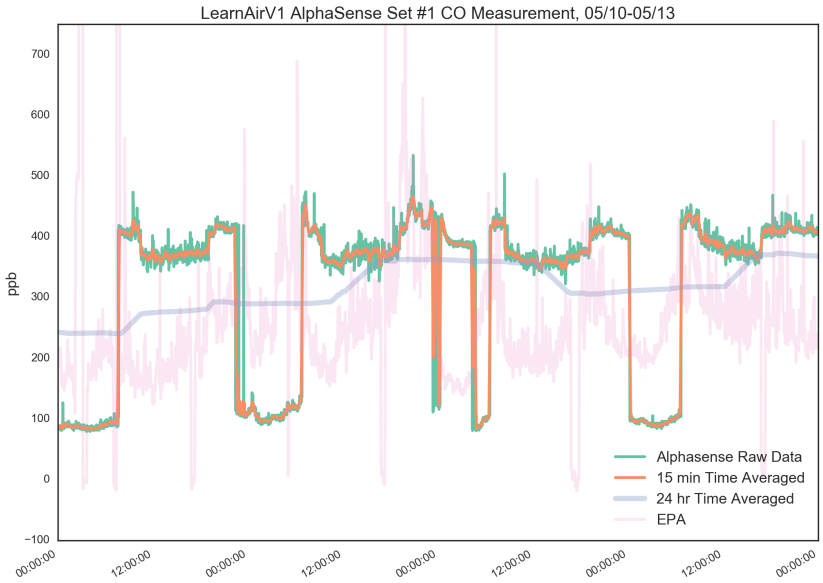


Figure 53: AlphaSense CO Sensor 1 Raw Data Zoomed

Machine Learning

parameters = 'C':[0.001, 0.1, 10, 1000], 'penalty':('L1', 'L2')

===== best ROC_AUC score 0.894883095983

===== best params 'penalty': 'L1', 'C': 1000

here's text referencing the (Table 16).

here's text referencing the (Table 17).

parameters = 'C':[0.001, 0.1, 10, 1000], 'penalty':('L1', 'L2')

===== best $ROC_AUCscore$0.904107344914

===== best params 'penalty': 'L1', 'C': 1000

here's text referencing the (Table 32).

here's text referencing the (Table **??**).

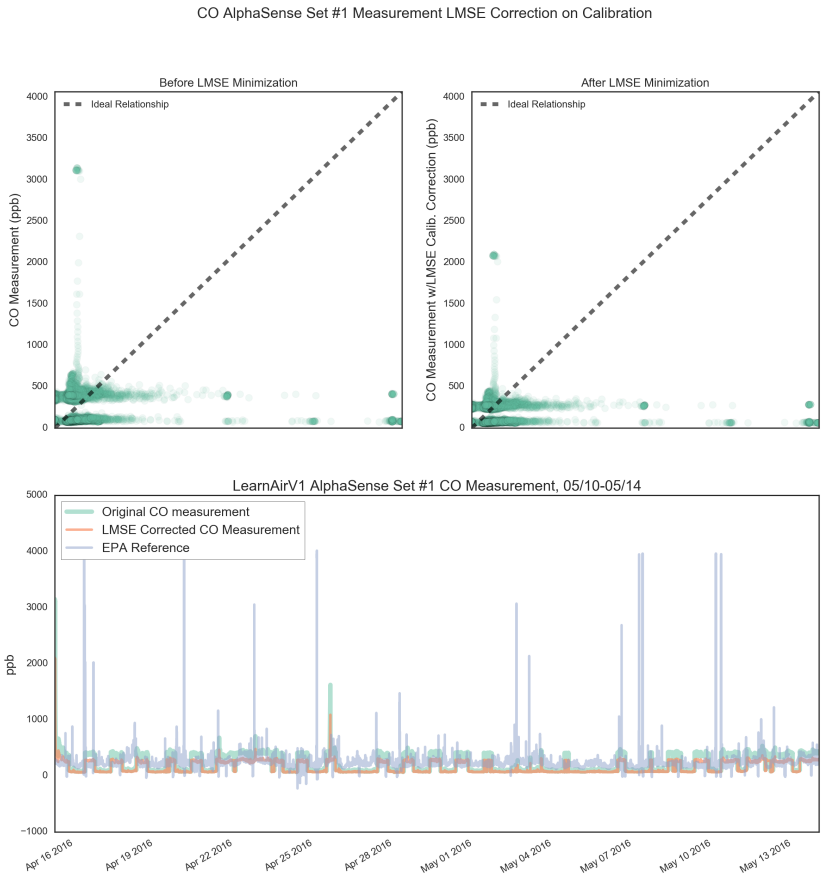


Figure 54: AlphaSense CO Sensor 1 after LMSE Calibration

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

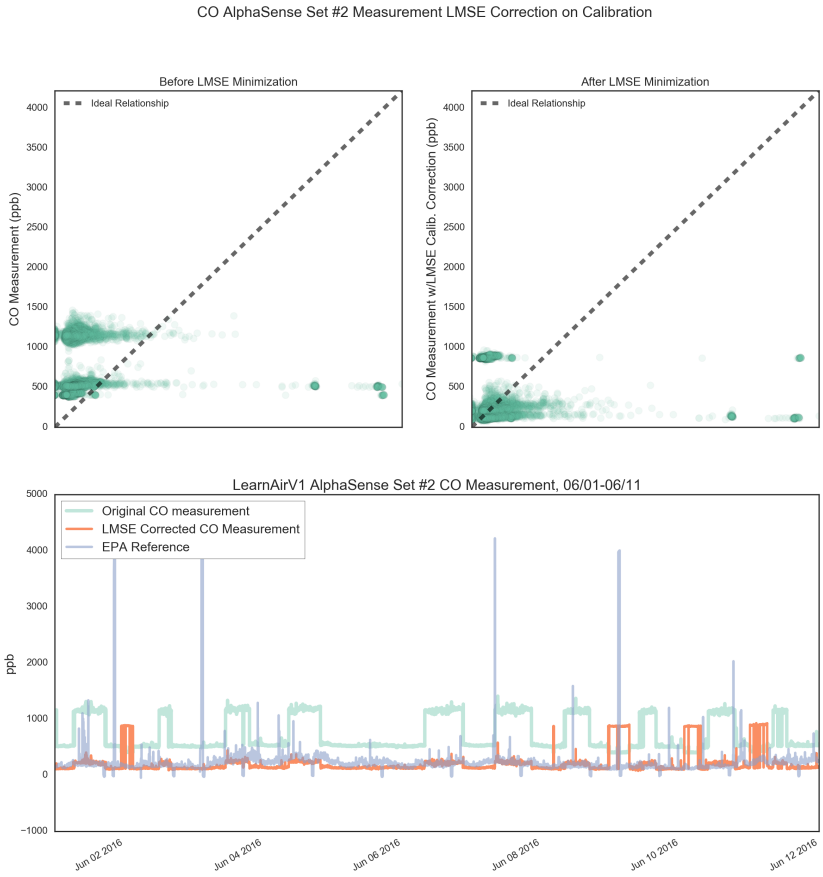


Figure 55: AlphaSense CO Sensor 2 after LMSE Calibration

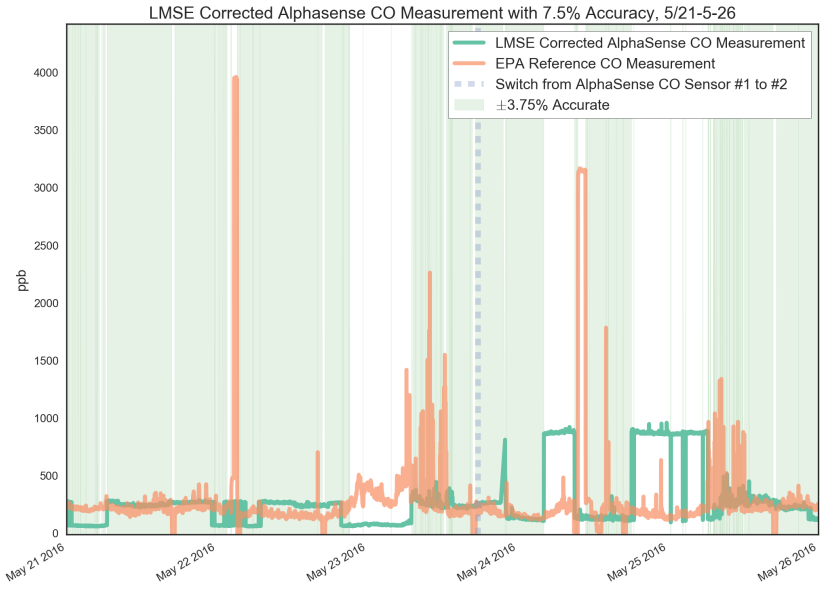


Figure 56: AlphaSense CO Sensor 1 and 2 with 7.5% Accuracy Threshold

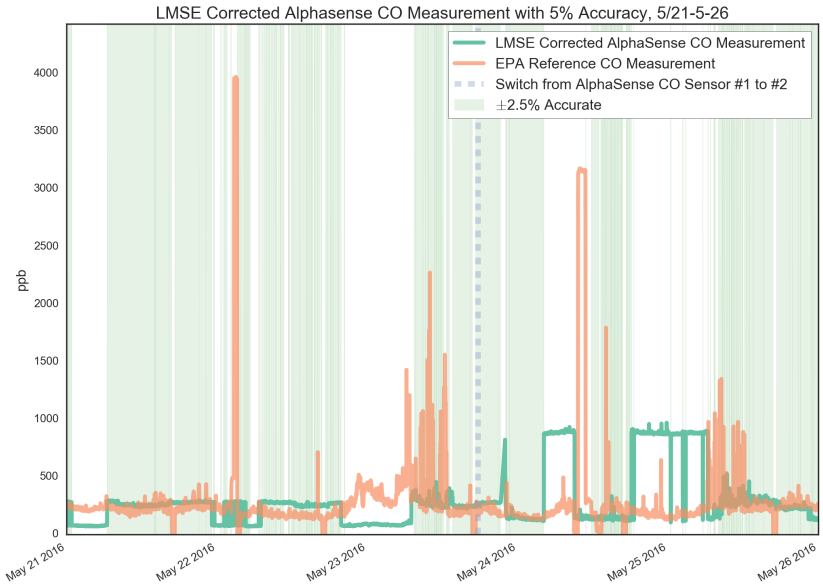


Figure 57: AlphaSense CO Sensor 1 and 2 with 5% Accuracy Threshold

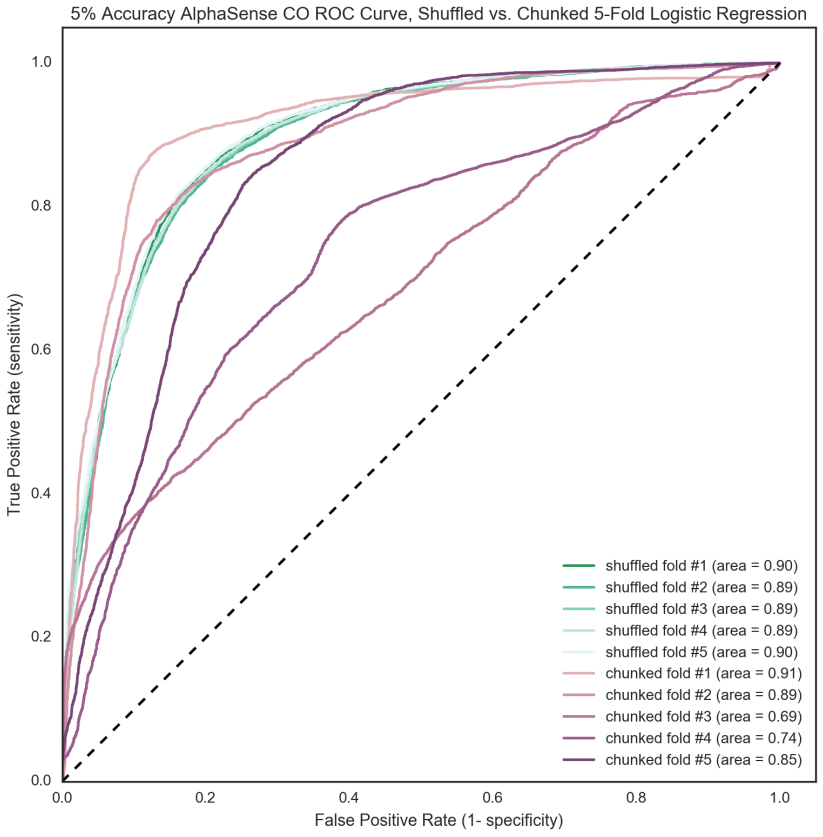


Figure 58: AlphaSense CO Sensor 1 ROC Curve

Table 15: Average AlphaSense CO Sensor 1 Confusion Matrix w/Shuffled K-Fold

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

Table 16: 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

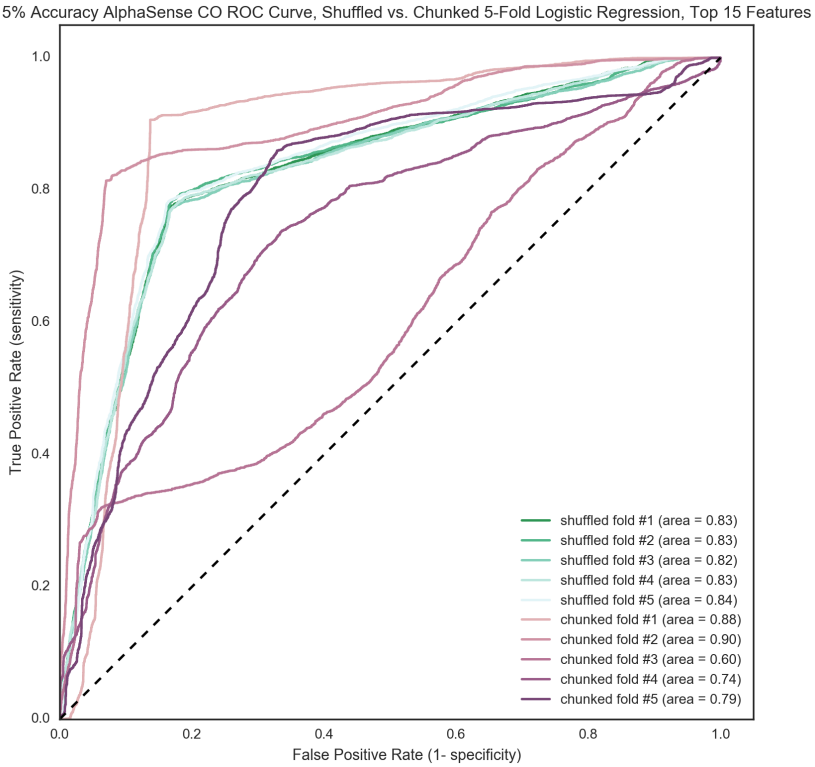


Figure 59: AlphaSense CO Sensor 1 ROC Using Top 15 Features

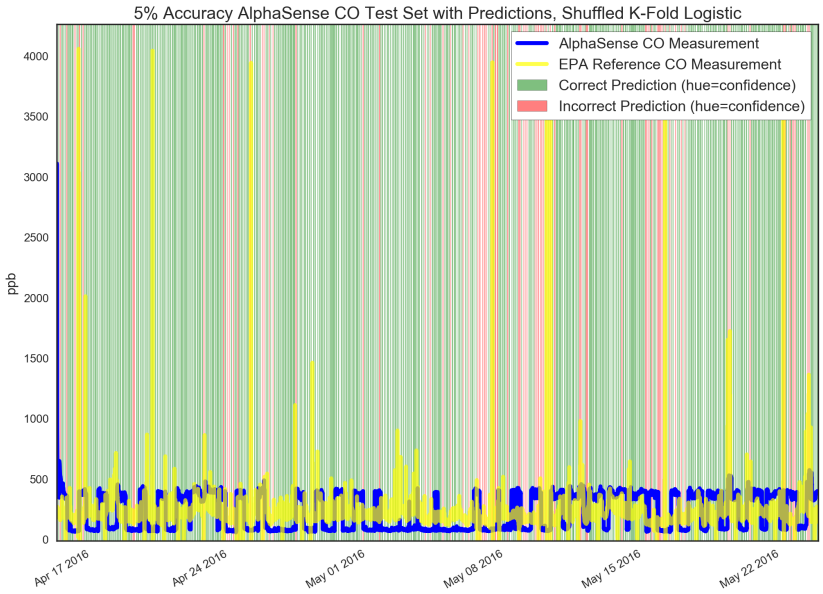


Figure 60: AlphaSense CO Sensor 1 Prediction Accuracy

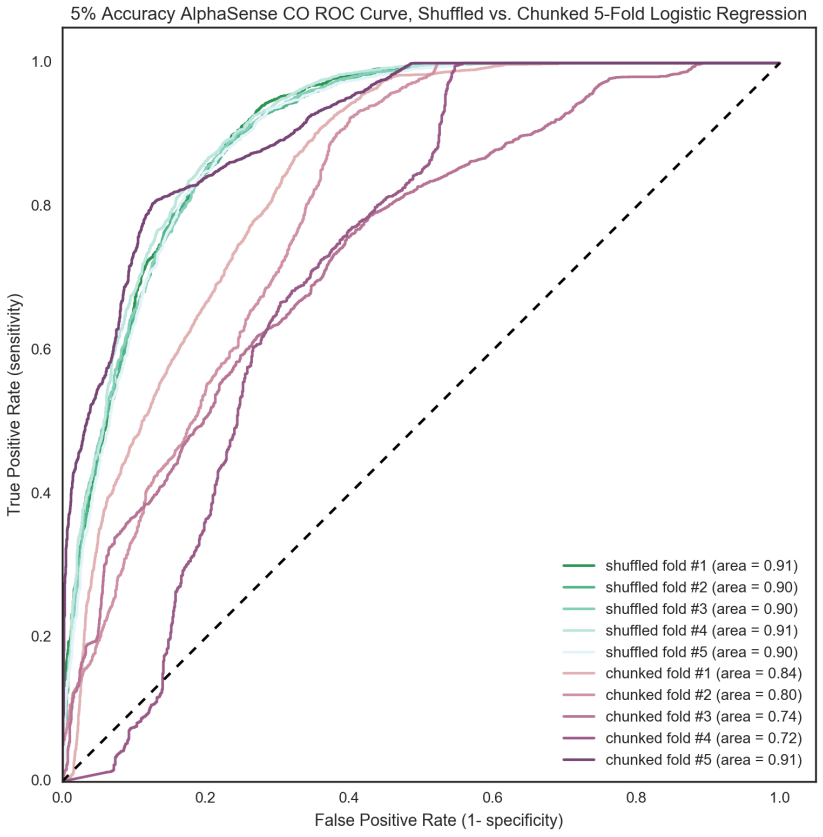


Figure 61: AlphaSense CO Sensor 2 ROC Curve

Table 17: 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

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

Table 19: Average AlphaSense CO Sensor 2 Confusion Matrix w/Shuffled K-Fold

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

Table 20: 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

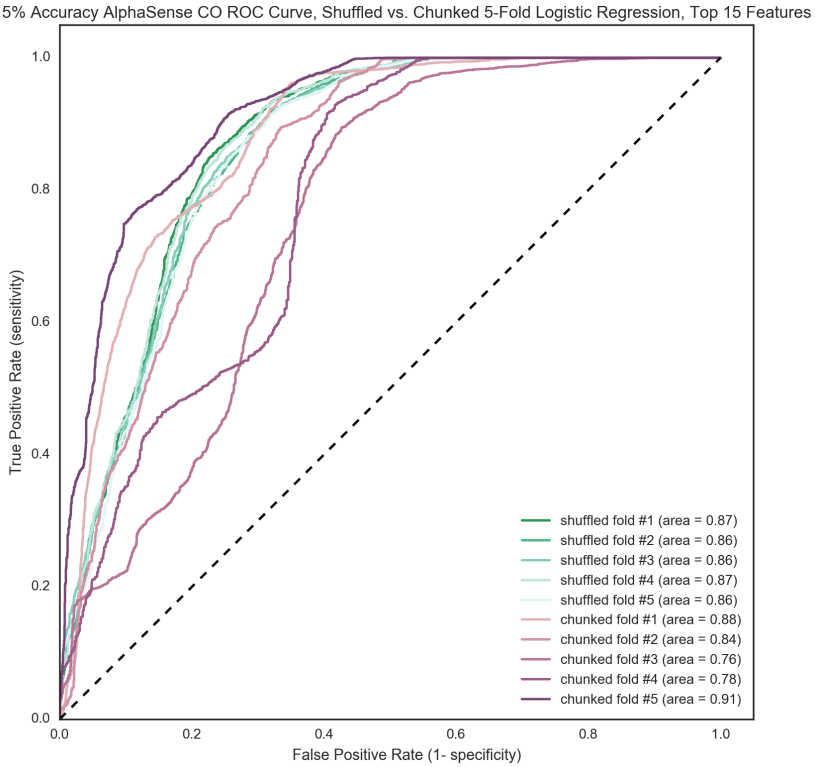


Figure 62: AlphaSense CO Sensor 2 ROC Using Top 15 Features

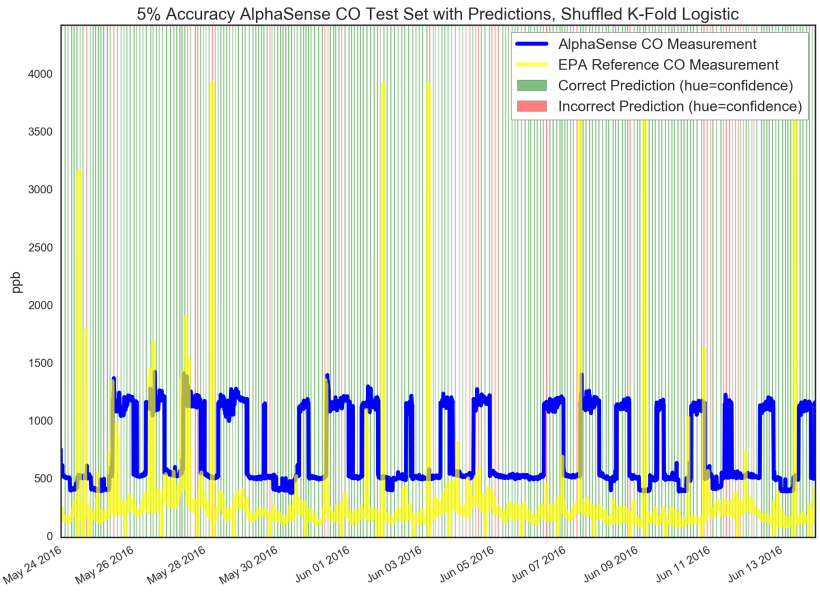


Figure 63: AlphaSense CO Sensor 2 Prediction Accuracy

AlphaSense NO2

This is about the alphasense NO2 results.

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

Talk about process for taking raw aux/working electrodes and making the basic calibration data.

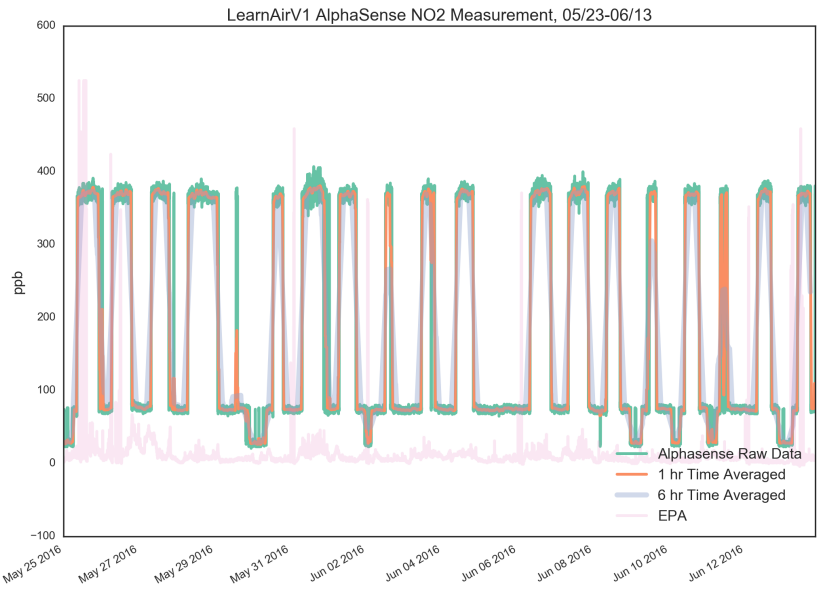


Figure 64: AlphaSense NO2 Raw Data

Machine Learning

parameters = 'C':[0.001, 0.1, 10, 1000], 'penalty':('L1', 'L2')

===== best ROC_AUC score 0.961586556822

===== best params 'penalty': 'L1', 'C': 1000

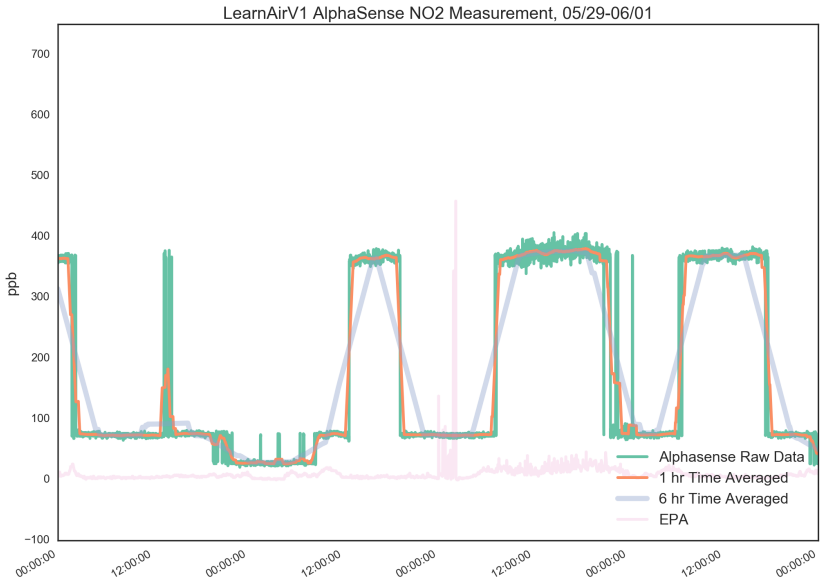


Figure 65: AlphaSense NO_2 Raw Data Zoomed

here's text referencing the (Table 24).

here's text referencing the (Table 25).

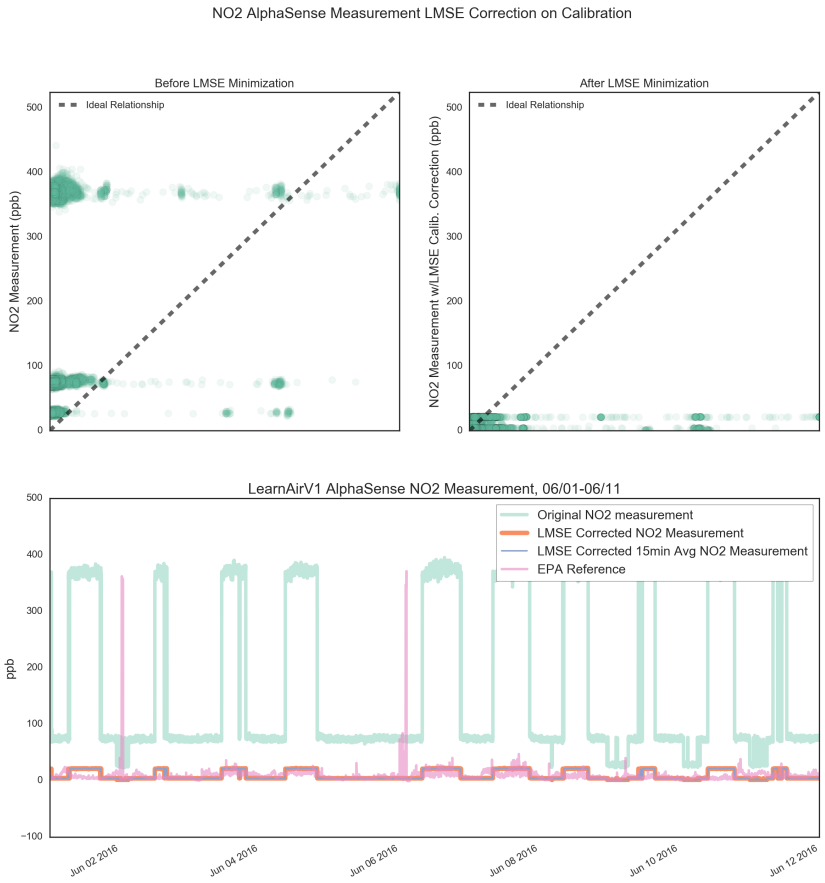


Figure 66: AlphaSense NO_2 after LMSE Calibration

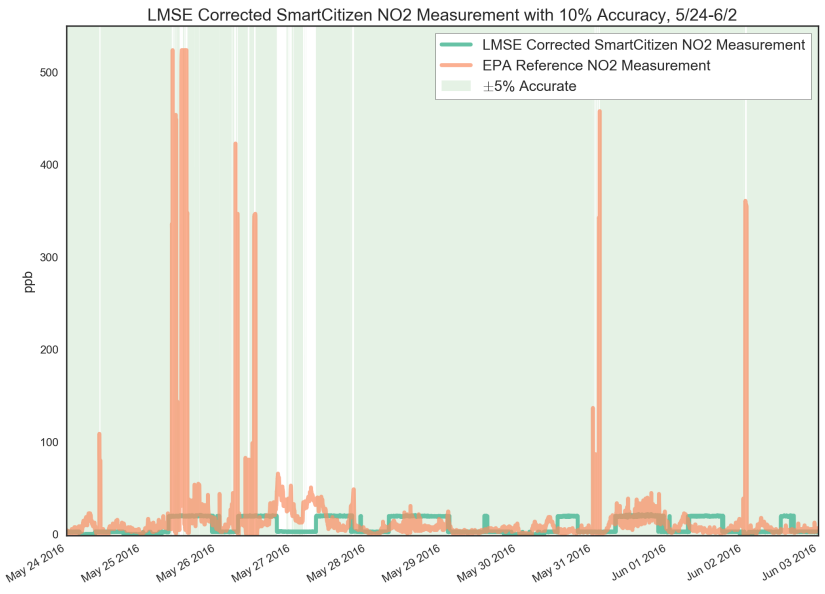


Figure 67: AlphaSense NO_2 with 10% Accuracy Threshold

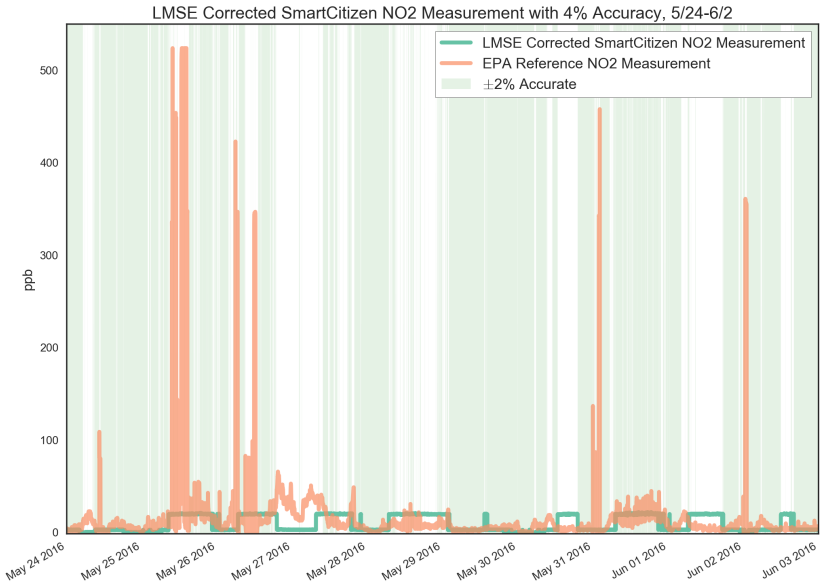


Figure 68: AlphaSense NO2 with 4% Accuracy Threshold

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 22: Error Rates for Predicting AlphaSense NO2 Accuracy with Logistic Regression

Table 23: Average AlphaSense NO2 Confusion Matrix w/Shuffled K-Fold

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

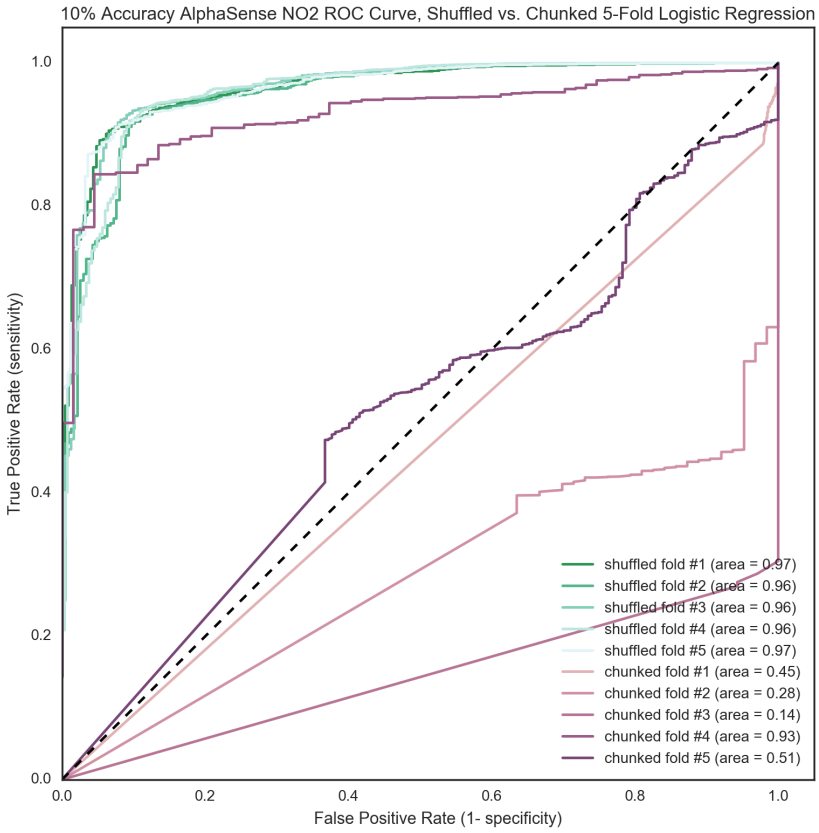


Figure 69: AlphaSense NO_2 ROC Curve

Table 24: 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

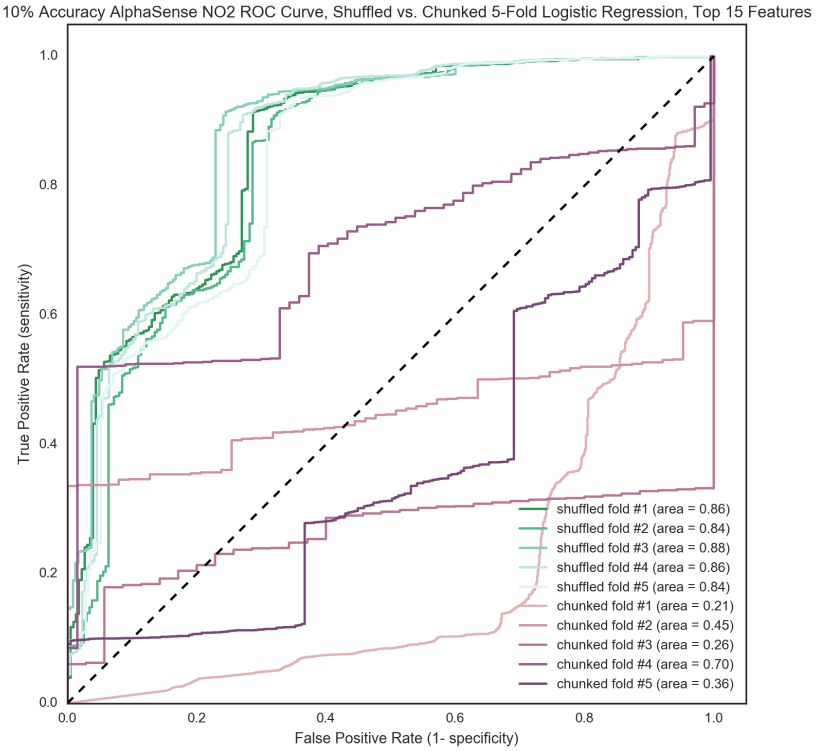


Figure 70: AlphaSense NO_2 ROC Using Top 15 Features

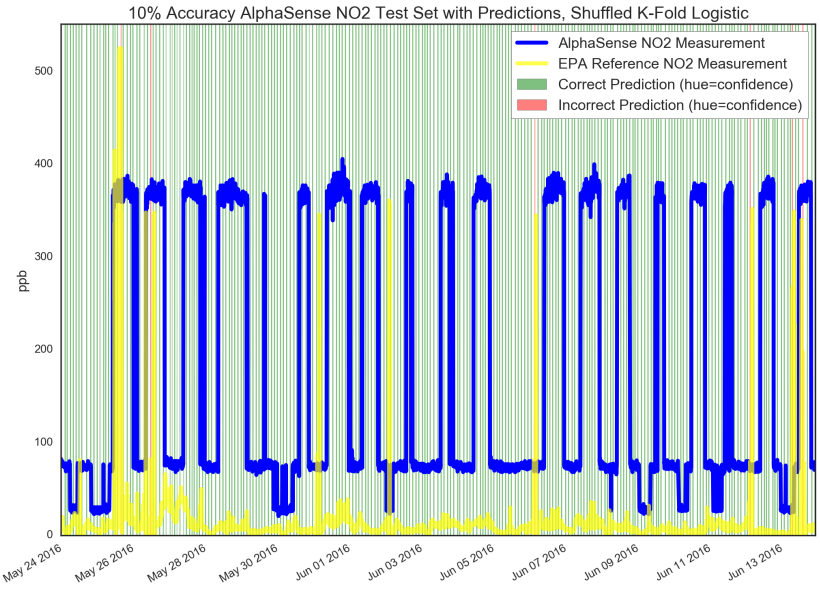


Figure 71: AlphaSense NO_2 Prediction Accuracy

Table 25: 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

This is about the alphasense O3 results.

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), that depends on the accuracy of the NO2 characterization, which complicates the calibration equation and introduces more opportunity for drift in the calibration process.

Pre-processing

Talk about process for taking raw aux/working electrodes and making the basic calibration data.

Old sensor, raw data with simple calibration based on datasheet.

New sensor all over the map with datasheet calibration because of NO2 calibration using raw calibration values and readings.

Machine Learning

parameters = 'C':[0.001, 0.1, 10, 1000], 'penalty':('L1', 'L2') , 2-Fold cross validation

===== best ROC_AUC score 0.725924836928

===== best params 'penalty': 'L1', 'C': 10

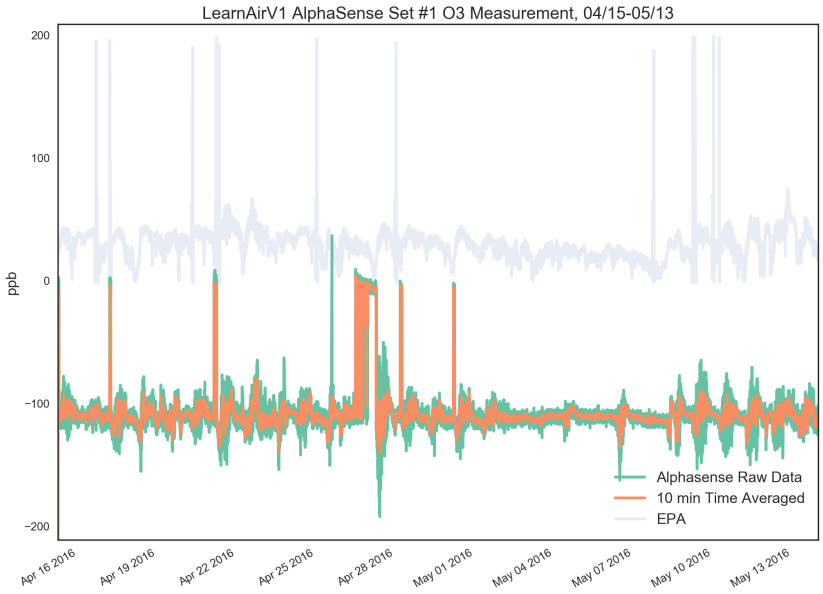


Figure 72: AlphaSense O_3 Sensor 1 Raw Data

here's text referencing the (Table 28).

here's text referencing the (Table 29).

parameters = 'C':[0.001, 0.1, 10, 1000], 'penalty':('L1', 'L2') , 2-Fold cross validation

===== best ROC_AUC score 0.816024265233

===== best params 'penalty': 'L1', 'C': 1000

here's text referencing the (Table 32).

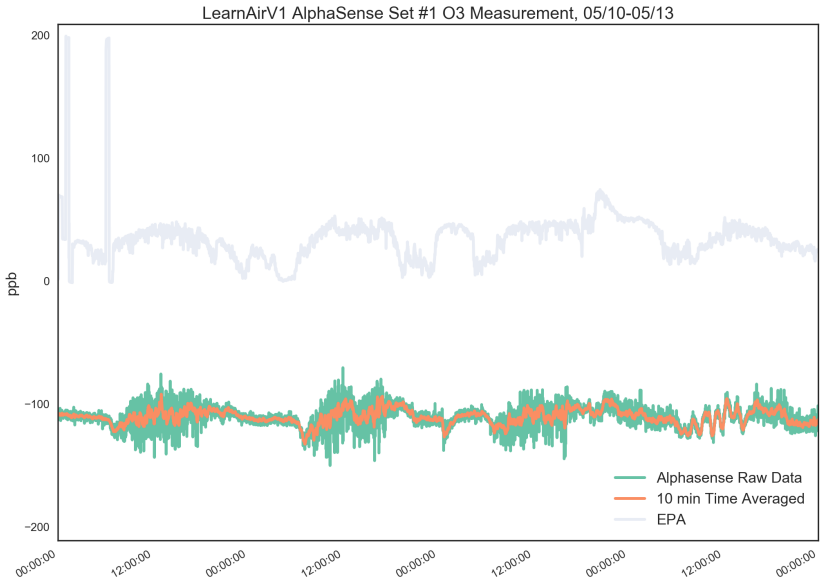


Figure 73: AlphaSense O_3 Sensor 1 Raw Data Zoomed

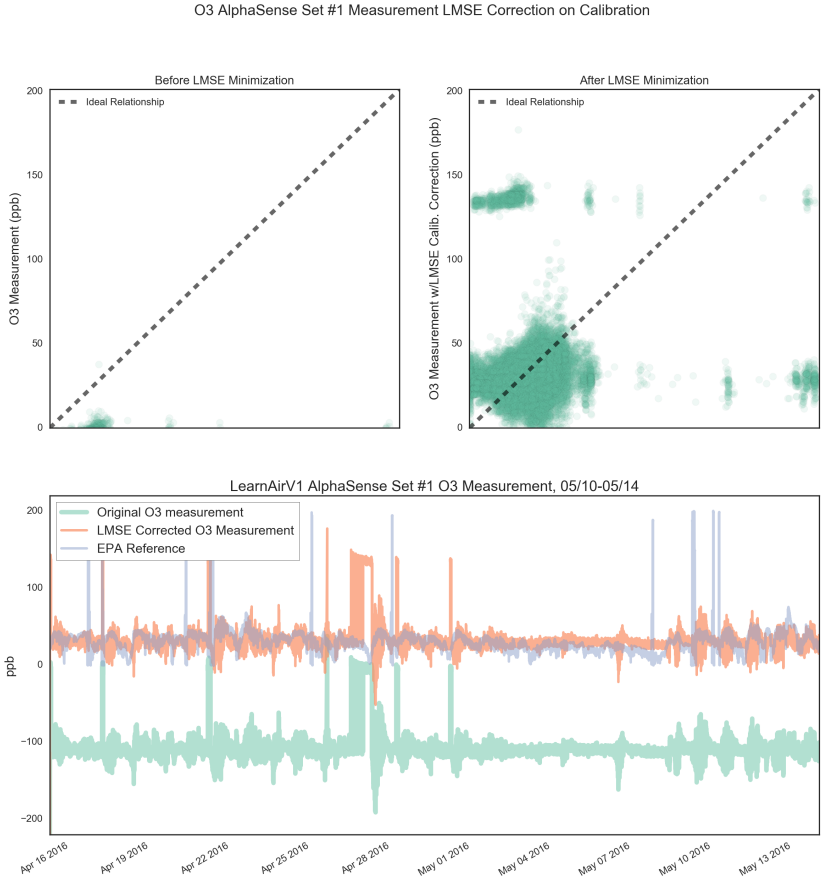


Figure 74: AlphaSense O_3 Sensor 1 after LMSE Calibration

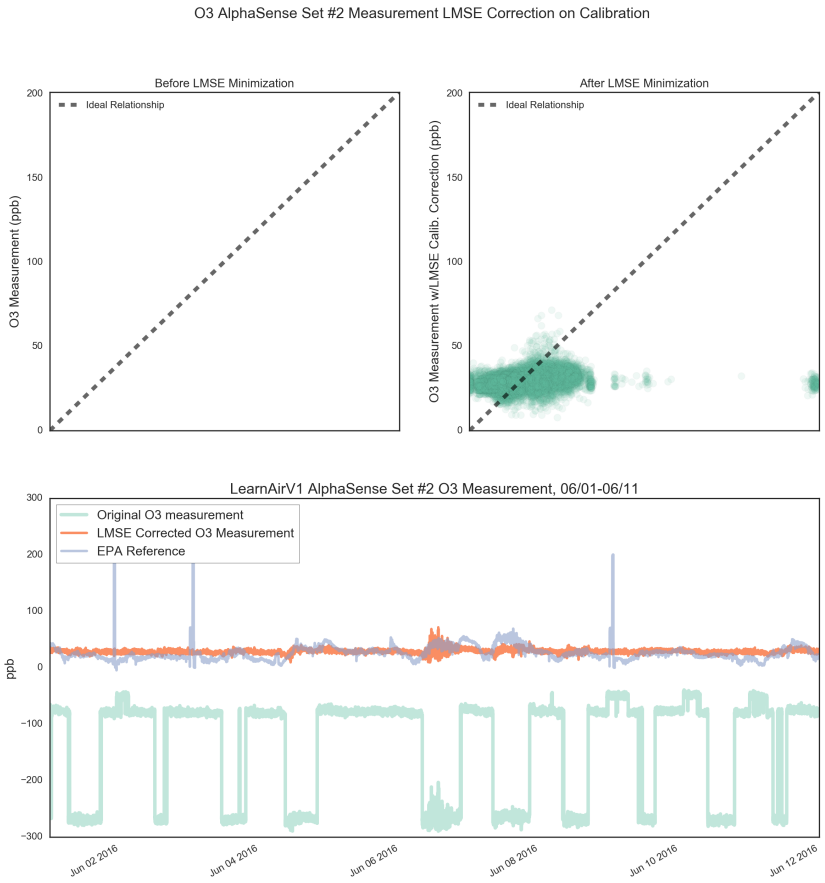


Figure 75: AlphaSense O3 Sensor 2 after LMSE Calibration

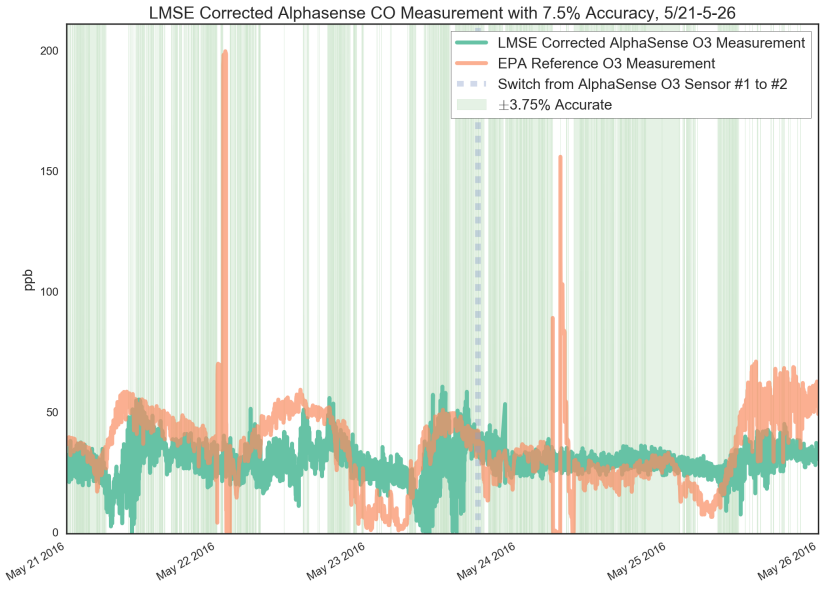


Figure 76: AlphaSense O3 Sensor 1 and 2 with 7.5% Accuracy Threshold

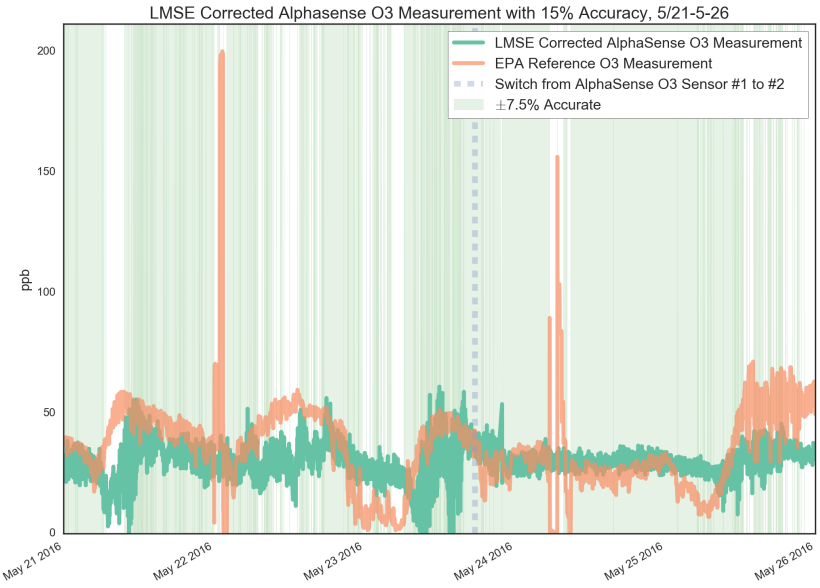


Figure 77: AlphaSense O3 Sensor 1 and 2 with 5% Accuracy Threshold

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

Table 27: AlphaSense O3 Sensor 1 Confusion Matrix w/Shuffled K-Fold

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

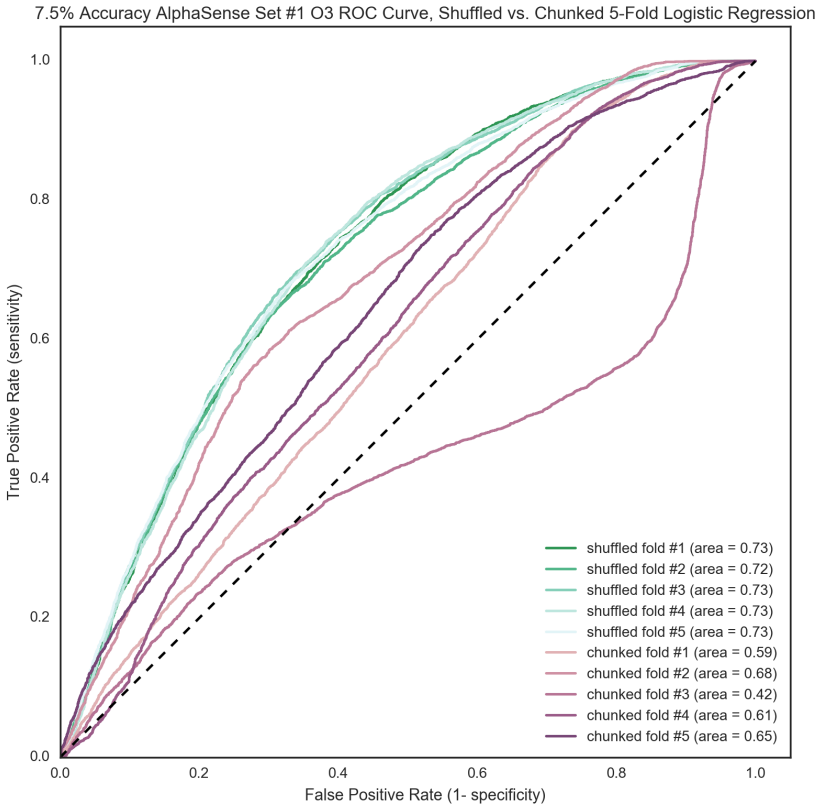


Figure 78: AlphaSense O3 Sensor 1 ROC Curve

Table 28: Top 15 Features from Random Forest for O3 Sensor 1, used in Pruned Logistic Regression

Feature	Importance
lmse_calib_as_o3	0.0388052728559
as_o3	0.038535780326
alphaS1_work	0.0256714422814
avg_10_as_o3	0.0143759967966
avg_10_lmse_calib_as_o3	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

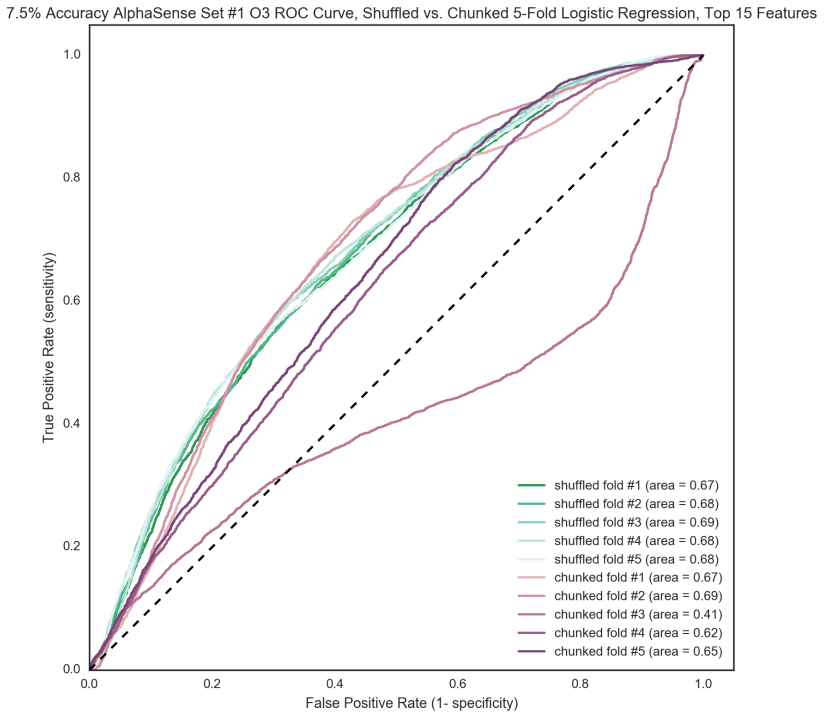


Figure 79: AlphaSense O3 Sensor 1 ROC Using Top 15 Features

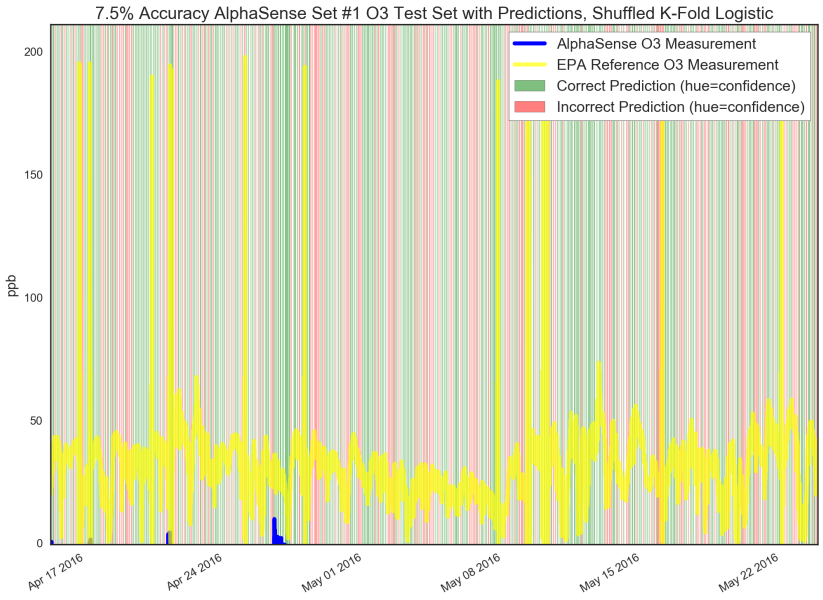


Figure 80: AlphaSense O3 Sensor 1 Prediction Accuracy

Table 29: 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

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

Table 31: AlphaSense O3 Sensor 2 Confusion Matrix w/Shuffled K-Fold

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

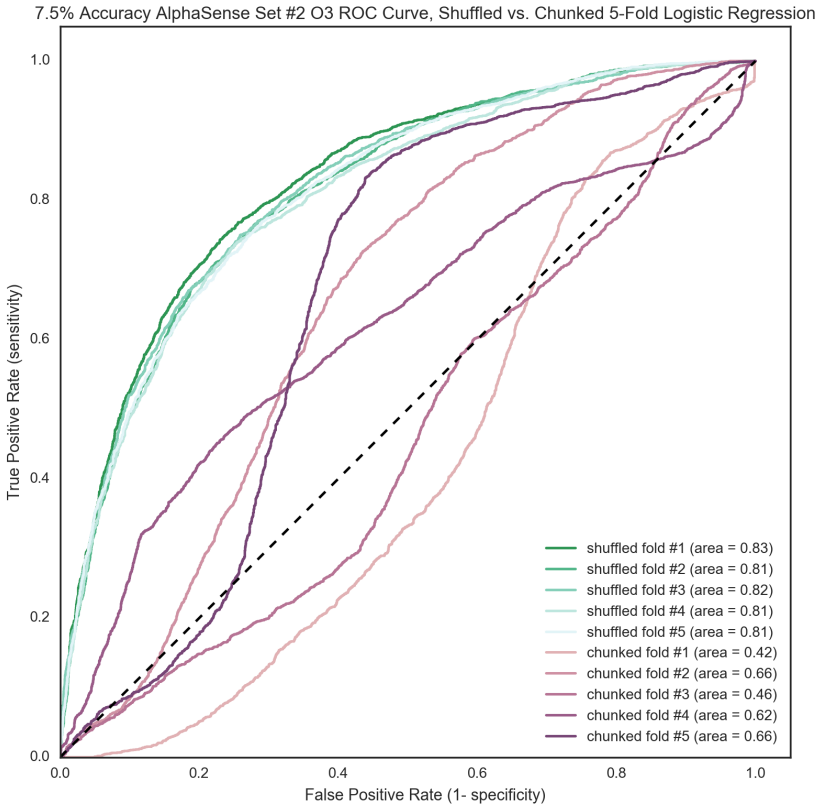


Figure 81: AlphaSense O3 Sensor 2 ROC Curve

Table 32: 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_o3	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

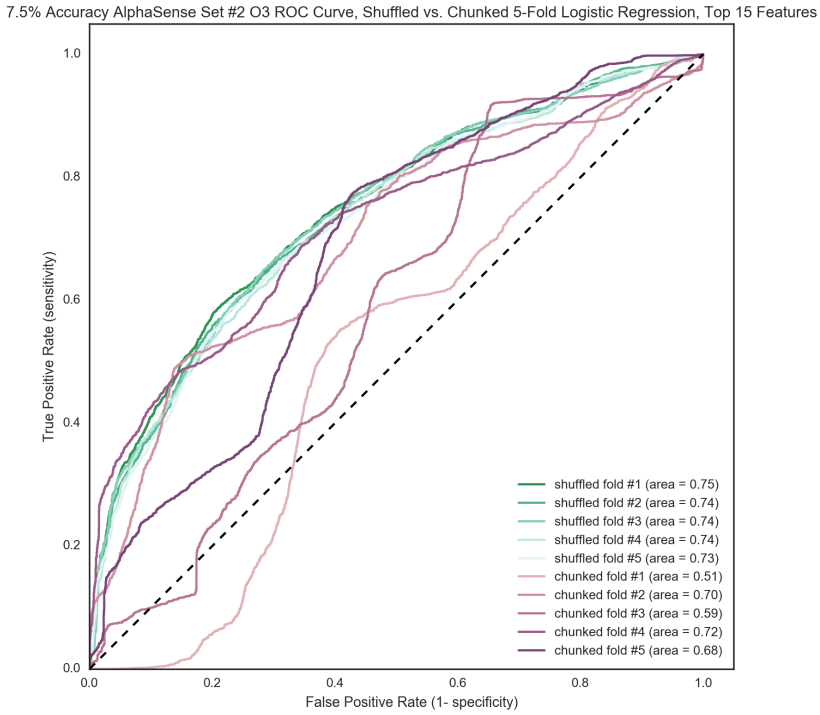


Figure 82: AlphaSense O3 Sensor 2 ROC Using Top 15 Features

here's text referencing the (Table 33).

Table 33: Top Features for Predicting AlphaSense O3 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

8. Conclusions and Future Work

The World Health Organization considers air quality to be the single leading environmental cause of death around the world. One in eight deaths worldwide are linked to poor air quality.

Applications

-example and provocation to citizen sensing community. -datastore for big data sharing. -extensible for any type of algorithm, particularly designed for learning algorithms to intelligently calibrate and predict accuracy. -EPA data sharing. Data sharing in general. -EPA testing of cheap sensors. -towards personal tracking, intelligent systems. highly scalable. -sell as platform/stamp of approval, change/define a new validation process for new sensors. all new consumer sensors push data to chain and are independently verified. Verified for regions of the country, verified for conditions, verified with a ranking of accuracy not just a yes or no.

Insights

-insight into shuffled vs. folded data, whether you've tested long enough in one climate. -insights into how these things break down - insights into feature reduction - explore how to build a cheap system, look at how strong a relationship is, and if it breaks in a statistically reliable way, inform our decisions moving forward. Is it corroborated across multiple feature reduction methods? -insight into feature reduction - explore the underlying mechanisms that might break a system down -insight into speed of sensor reaction, and what that means for building a system like this, how it interacts with health.

-insight into cheap sensors, contrast SQMD with my results. -insight into windspeed. -insight into data sharing and backend design. -usefulness of accuracy of predictions? not large now. Would have to be statistical techniques. -insight into overall system feasibility.

Future Work

as with any good project, this thesis begets as many questions as it answers. -wind sensing. -pollen, scrape. no api. realtime Traffic data a la google maps. Traffic type data based on road type (estimate of heavy deasel etc). Construction. Map and distance from a street, buildings, etc. Age of infrastructure. SES of neighborhood as a predictor of car age and emissions. Aeris. -mount on Aclima car. -more intelligent input data - distance to road, vibration, speed traveling, traffic pattern data. -approached by a few people will to test. -more data, test with newer sensors. -try other machine learning techniques, time-dependent, deep-learning. all possible. -data backend infrastructure buildout, promote in community. -more tools for backend crawling. -help educate communities. -help automate testing for big organizations. -build out recommedation system for devices that perform well in different environments. -build out trustworthiness system for devices based on make/model to characterize interpart variability. -statistical tools for combining sensor data - spikes missing? is there a way to predict frequency/intensity in an area by looking at closest high quality sensor?

Summary

-hand waving goal - a sensor that learns when it's around a better sensor what conditions it can be trusted. As you carry it away from that sensor, it predicts what measurements it makes are trustworthy, and what measurements are not.

we set out to achieve x. we did. This shows promise, this was lacking. many future directions.

Did (1). Did (2). Did (3). Steps toward an interesting system.

Right now this is immediately useful for automatic calibration, and for showing systemic errors and correlates. There needs to be a hu-

man in the loop that looks and understands the physics right now, but with enough training and another layer of abstraction it may be possible for the machine to identify common issues (common cross-sensitivities, optical issues around fog or road dust, typical deficiencies for certain types of sensors to temperature/humidity/response time).

Appendix A - Notes on Project Selection and Prior Work

An aside about paths mistook.

Chambering air for mobile use, 'fixing'/modeling air flow issues through a device using machine learning

prototype built.

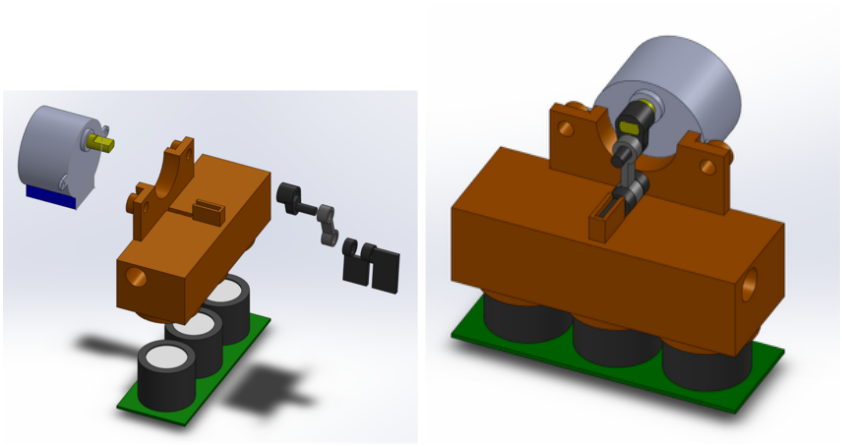


Figure 84: Original Concept #1

chambering and pressurizing the air slightly

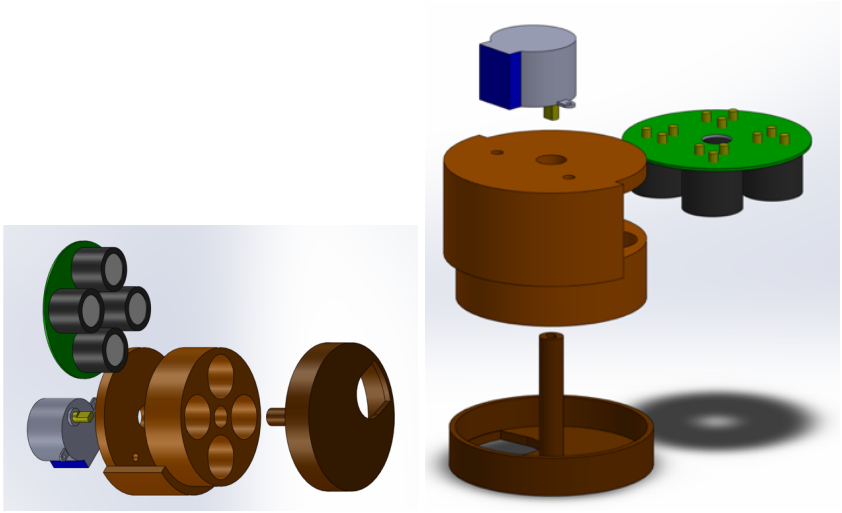
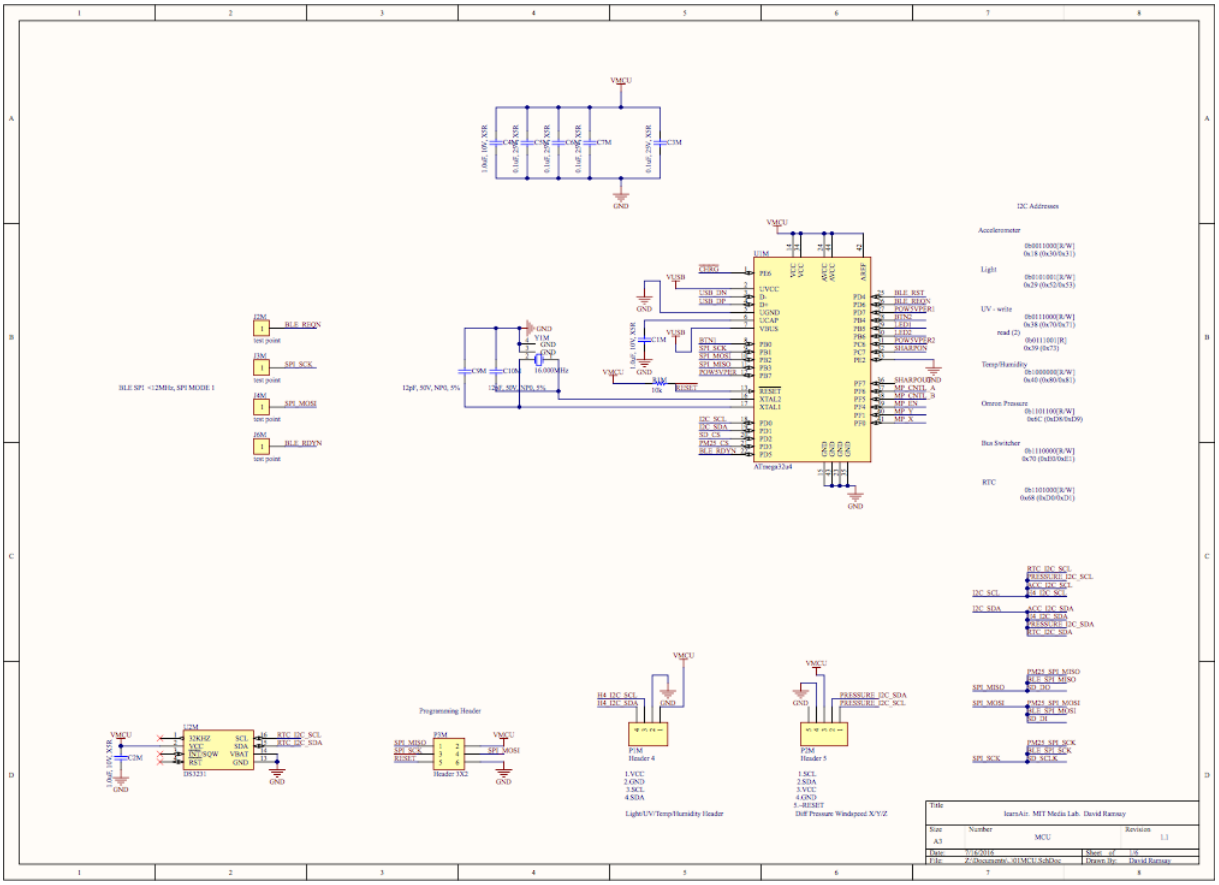


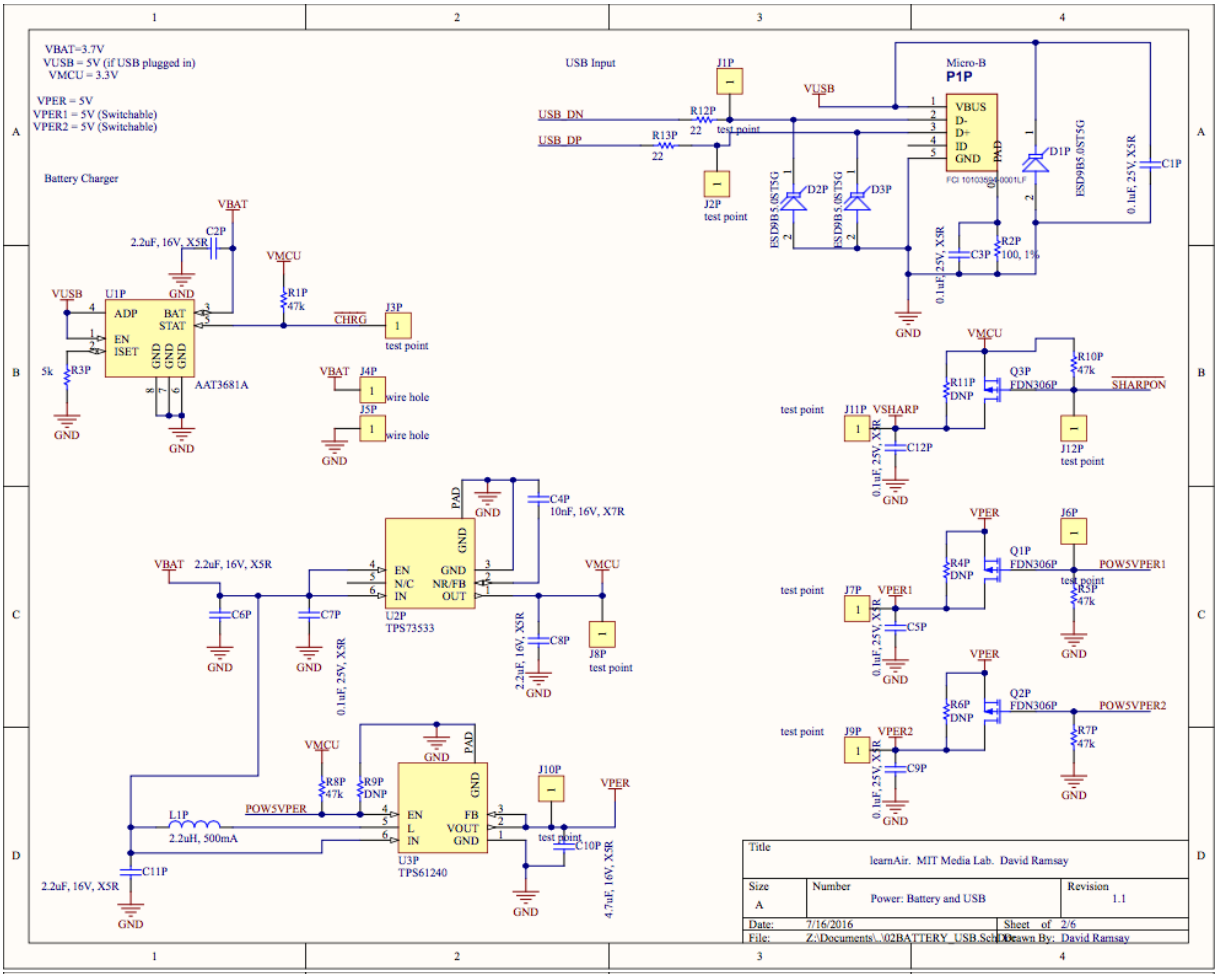
Figure 85: Original Concept #2

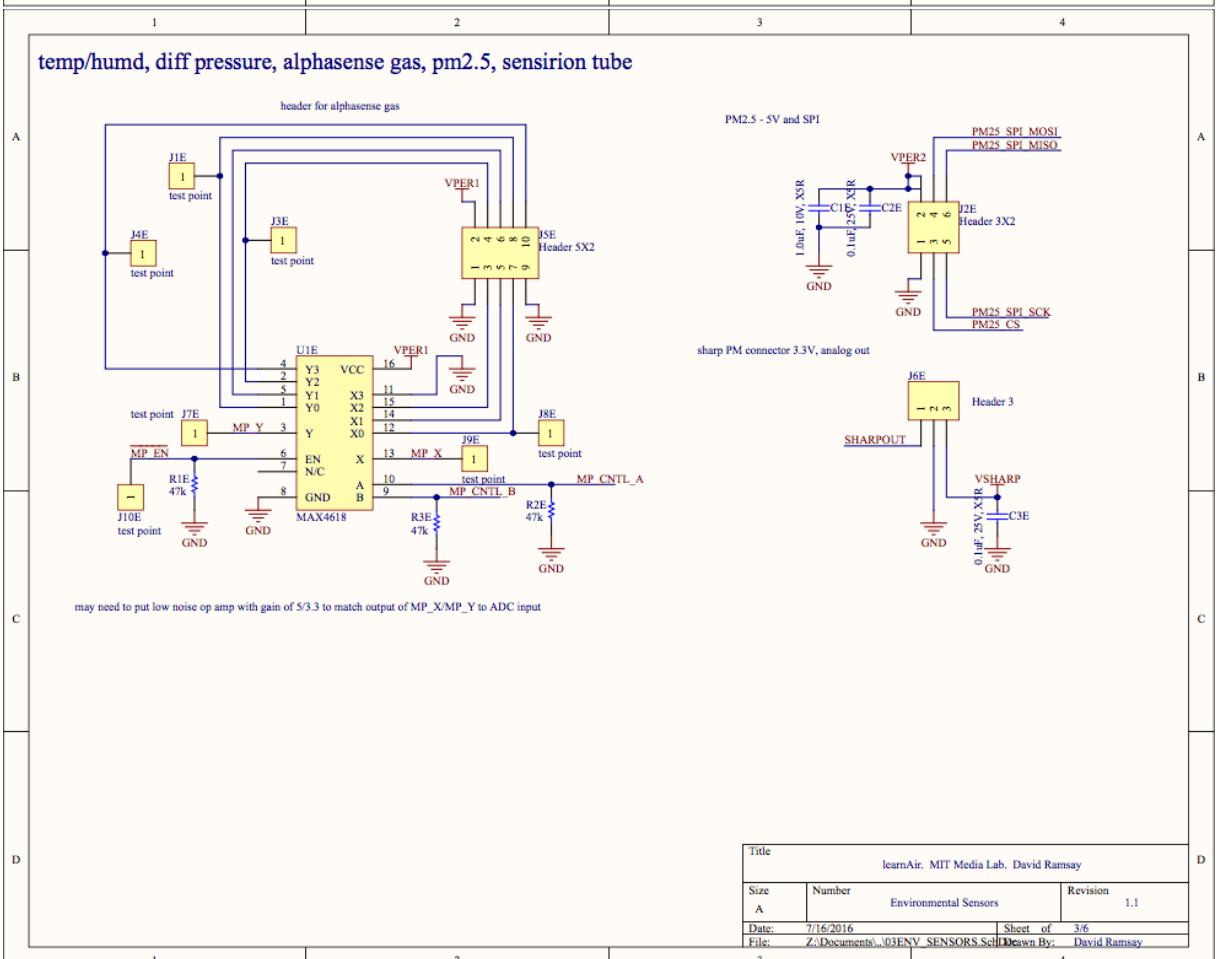
Appendix B - Hardware and Firmware

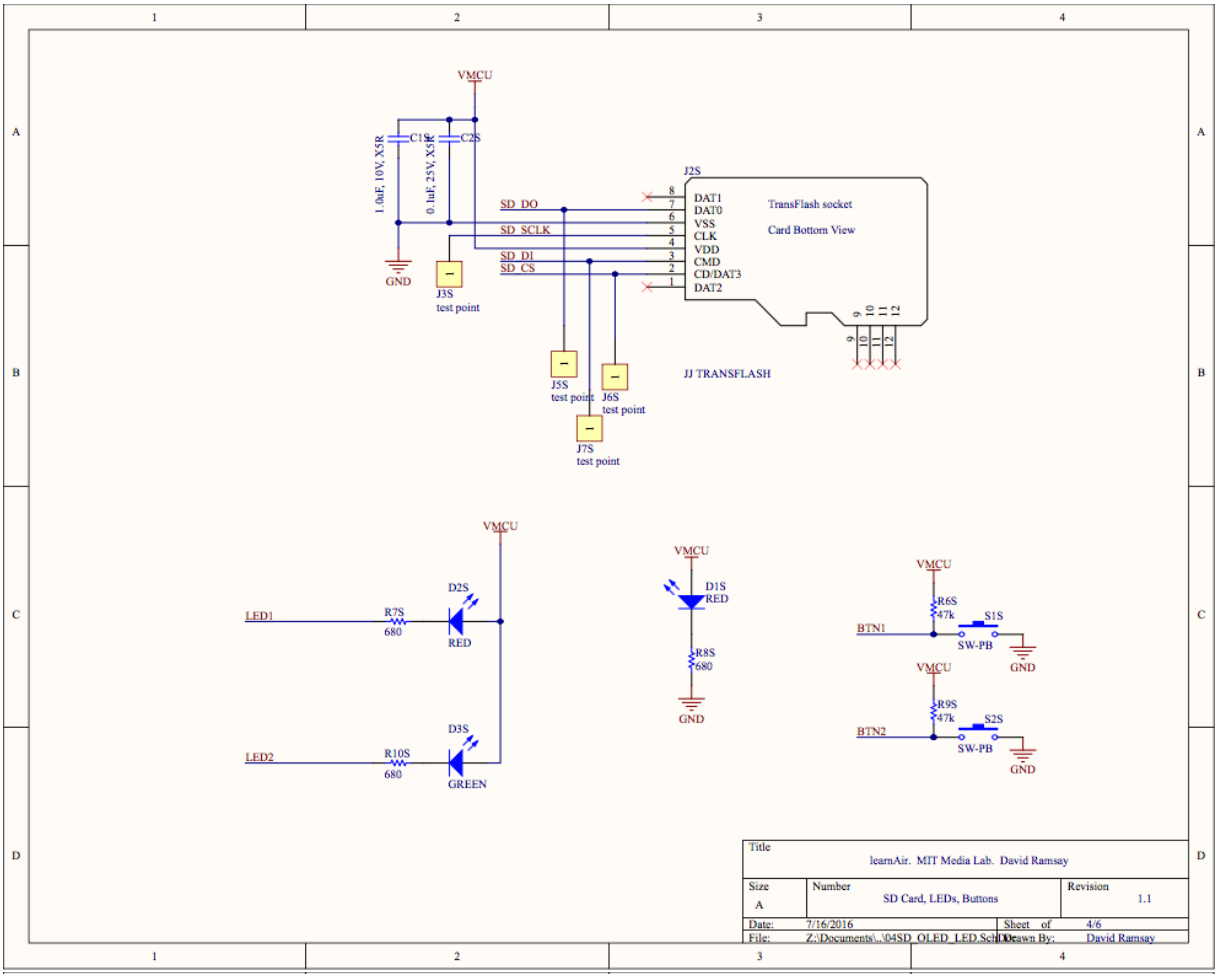
Schematics

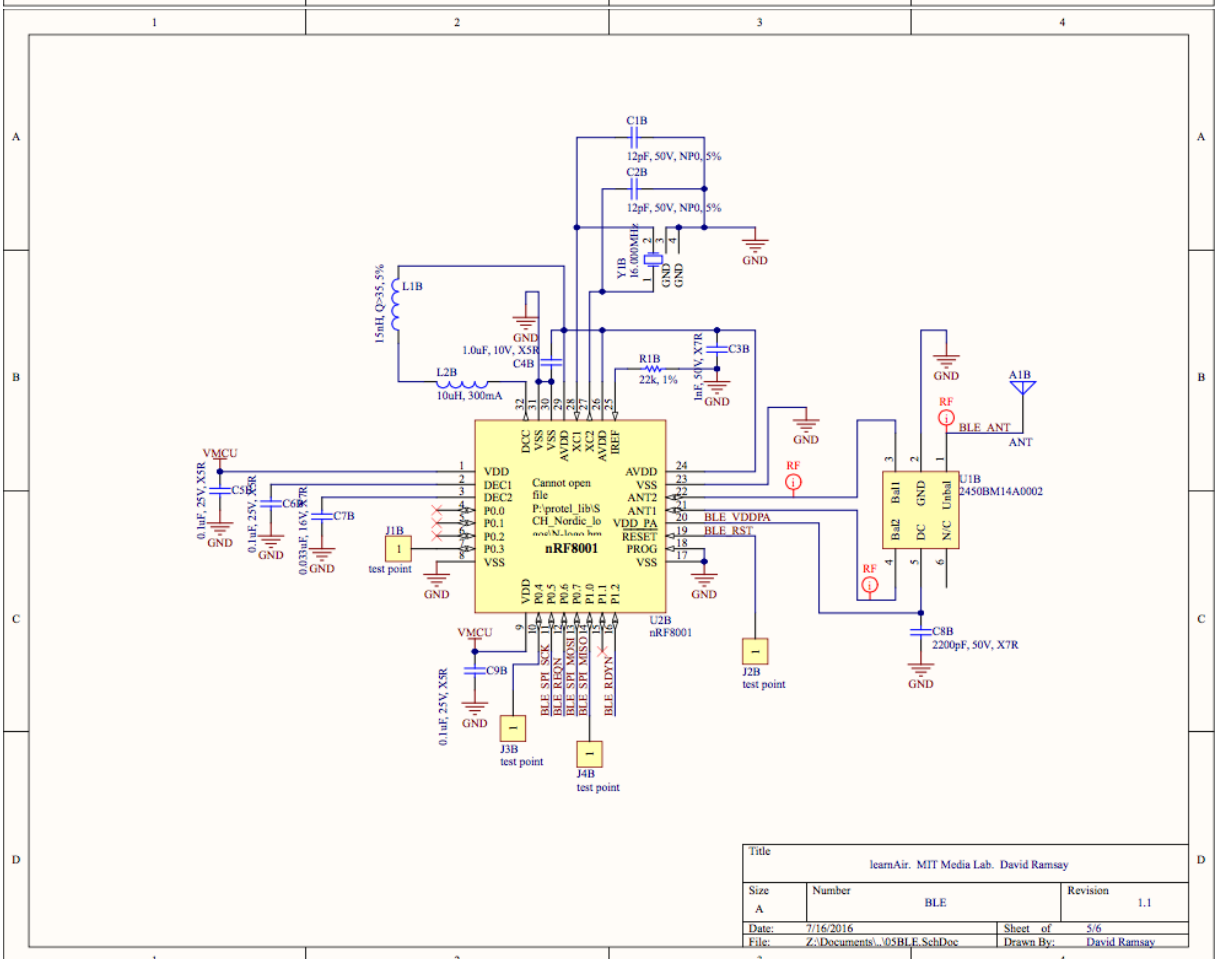
LearnAir V2 schematics are below.

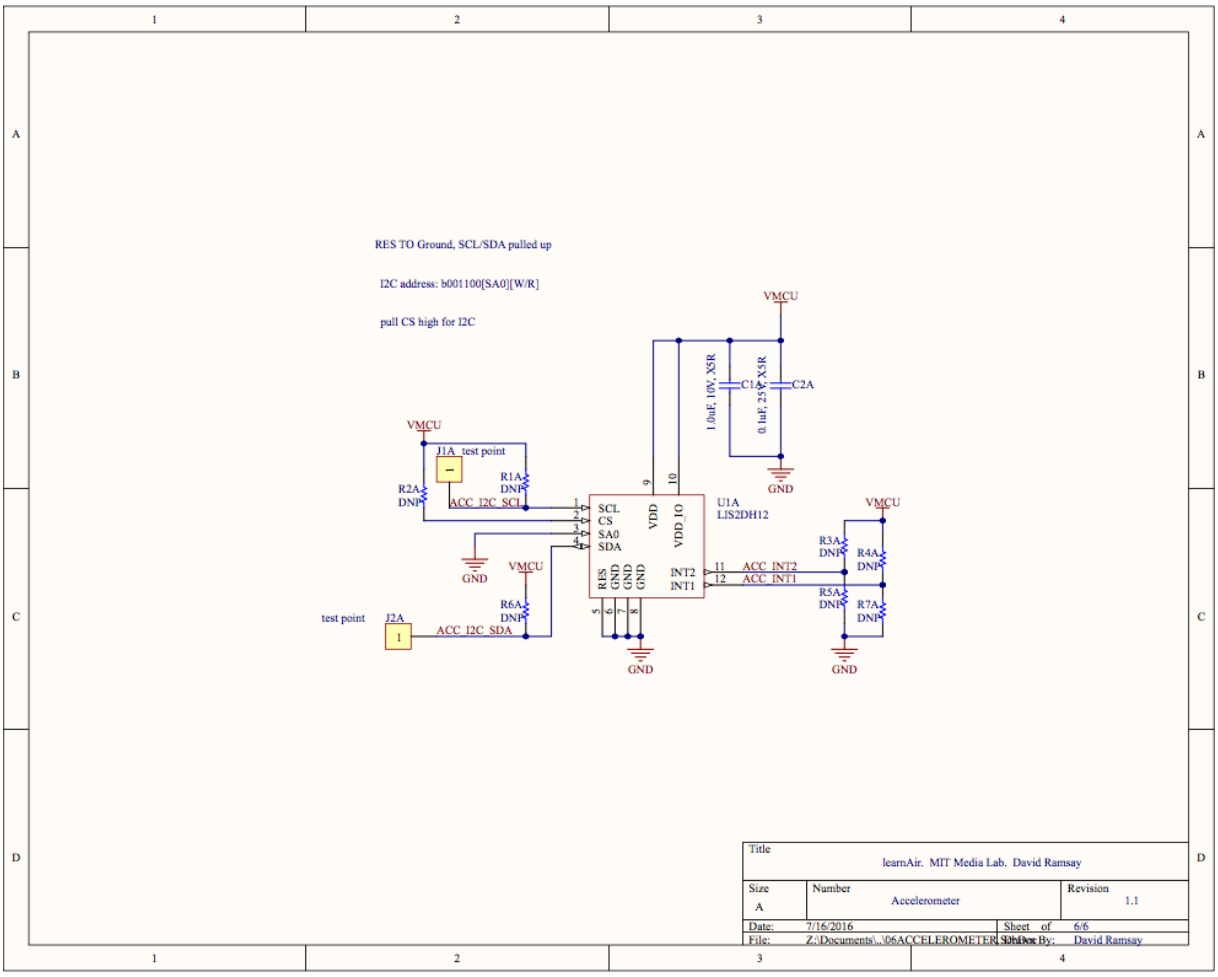



VBAT=3.7V
VUSB = 5V (if USB plugged in)
VMCU = 3.3V
VPER = 5V
VPER1 = 5V (Switchable)
VPER2 = 5V (Switchable)
Battery Charger
USB Input
Micro-B
P1P
AAT3681A
TPS73533
TPS61240
FDN306P
learnAir. MIT Media Lab. David Ramsay
Power: Battery and USB
Revision 1.1
Date: 7/16/2016
Sheet of 2/6
File: Z:\Documents\..\02BATTERY_USB.SchDoc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

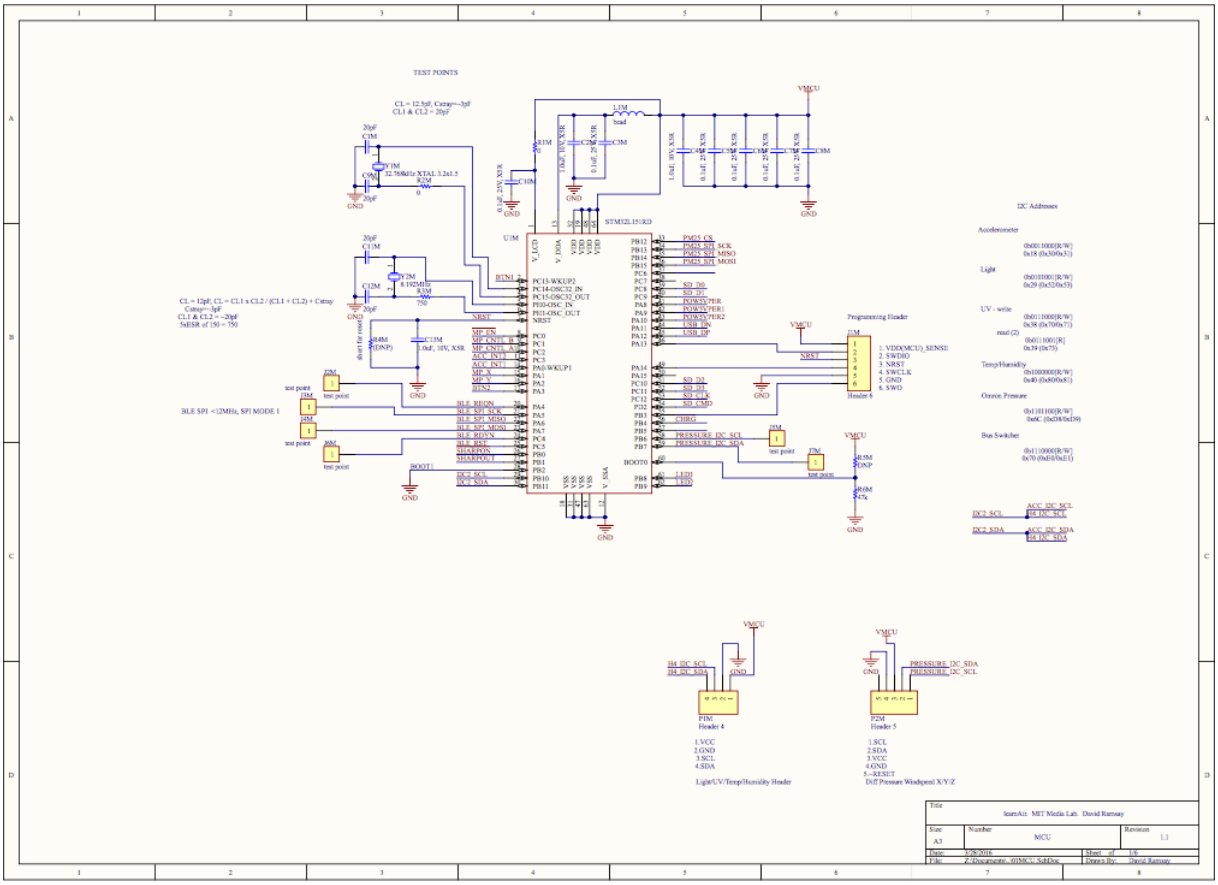

temp/humd, diff pressure, alphasense gas, pm2.5, sensirion tube
header for alphasense gas
PM2.5 - 5V and SPI
sharp PM connector 3.3V, analog out
MAX4618
may need to put low noise op amp with gain of 5/3.3 to match output of MP_X/MP_Y to ADC input
learnAir. MIT Media Lab. David Ramsay
Environmental Sensors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

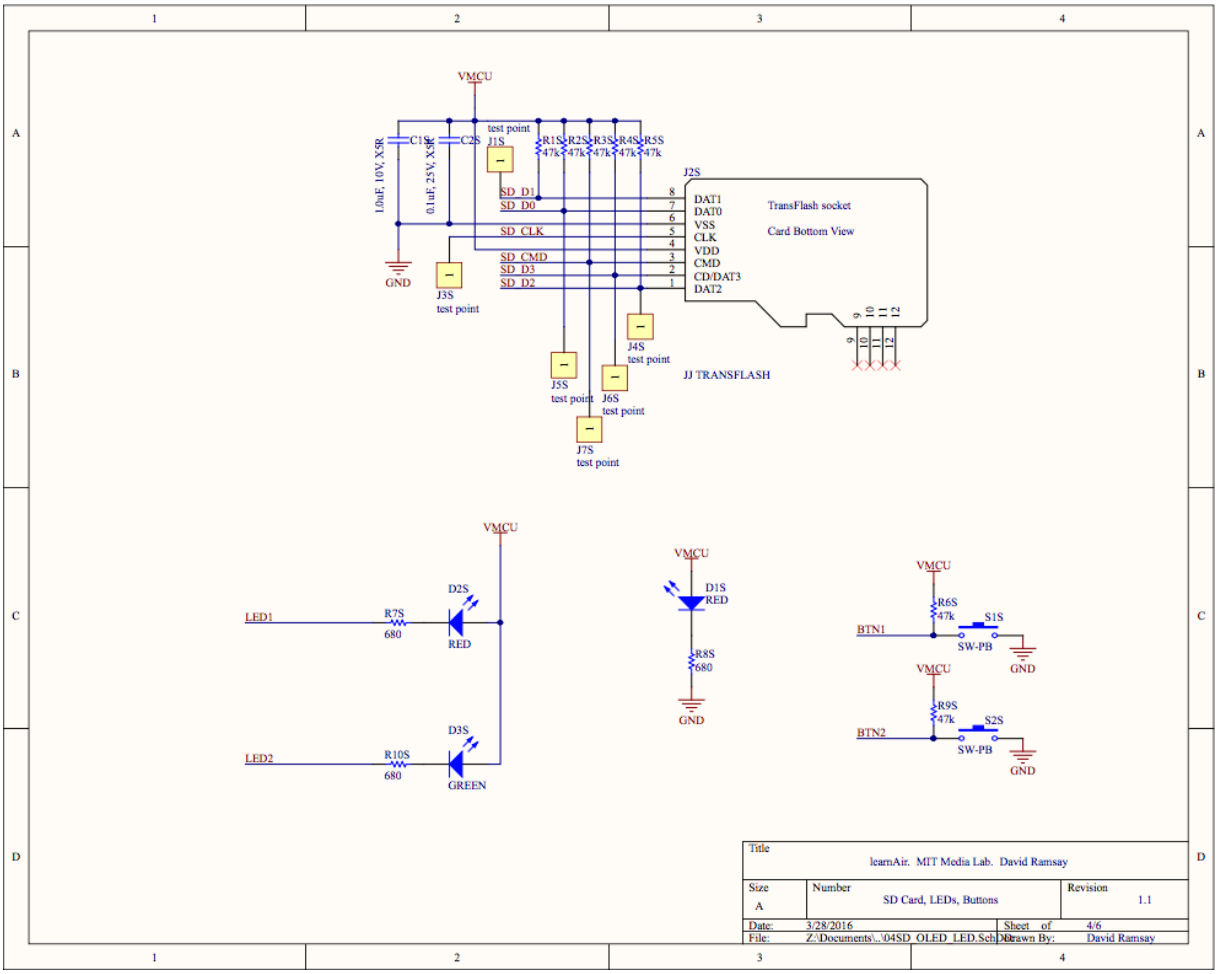

VMCU
C1S
1.0uF, 10V, X5R
C2S
0.1uF, 25V, X5R
GND
J3S
test point
J2S
DAT1
DAT0
VSS
CLK
VDD
CMD
CD/DAT3
DAT2
TransFlash socket
Card Bottom View
SD_DO
SD_SCLK
SD_DI
SD_CS
J5S
test point
J6S
test point
J7S
test point
JJ TRANSFLASH
LED1
R7S
680
D2S
RED
LED2
R10S
680
D3S
GREEN
D1S
RED
R8S
680
BTN1
R6S
47k
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
BLE_SPI_SCK
BLE_RDYN
BLE_SPI_MISO
BLE_SPI_MOSI
BLE_RDYN
C9B
0.1uF, 25V, X5R
J3B
test point
J4B
test point
J2B
test point
A1B
ANT
BLE_ANT
U1B
2450BM14A0002
C8B
2200pF, 50V, X7R
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
C1A
C2A
1.0uF, 10V, X5R
0.1uF, 25V, X5R
GND
J1A test point
R1A DNP
R2A DNP
ACC_I2C_SCL
U1A LIS2DH12
SCL
CS
SA0
SDA
VDD
VDD_IO
RES
GND
INT2
INT1
ACC_INT2
ACC_INT1
R3A DNP
R4A DNP
R5A DNP
R7A DNP
R6A DNP
test point
J2A
ACC_I2C_SDA
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

Hardware Analysis

Figure 86 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 87 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

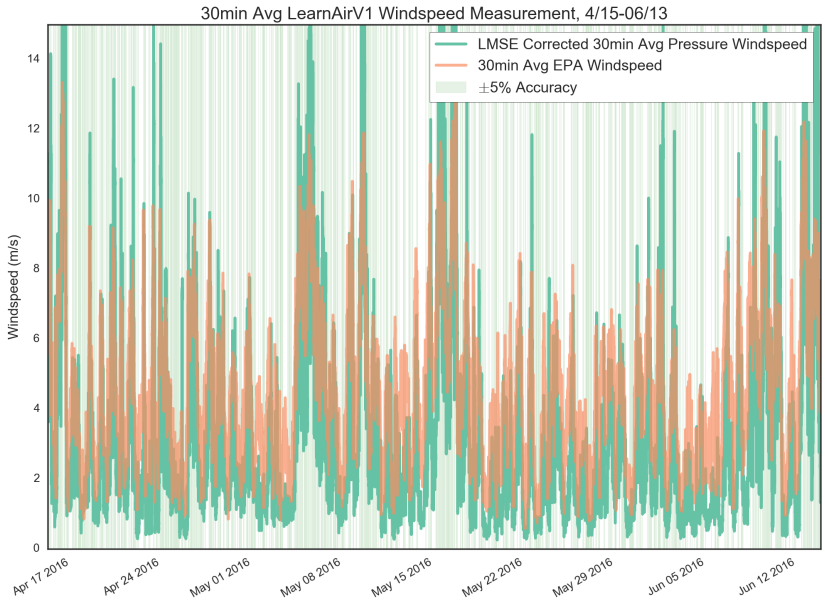


Figure 86: Wind Speed Measurement with 10% Accuracy

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.

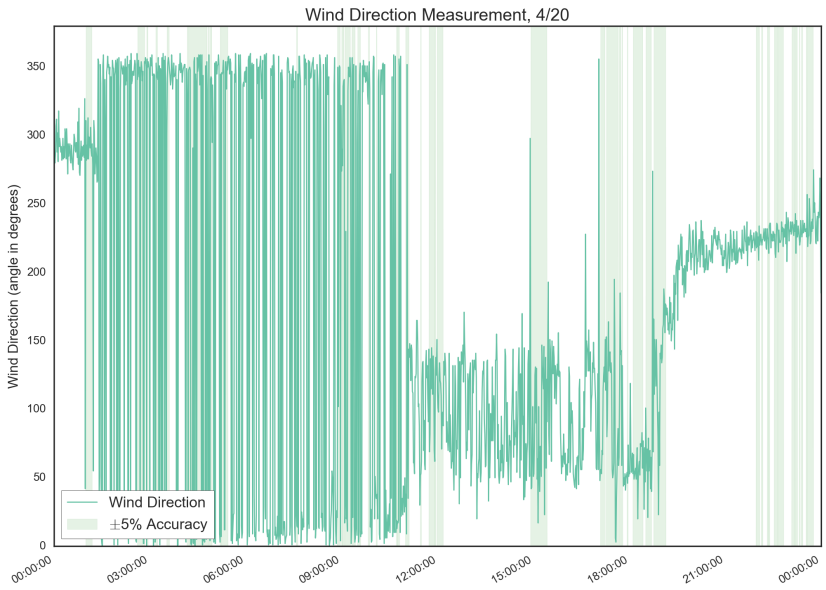


Figure 87: Wind Direction with 10% Accuracy WindSpeed Measurements Denoted

Appendix C - ChainAPI Code

??

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.

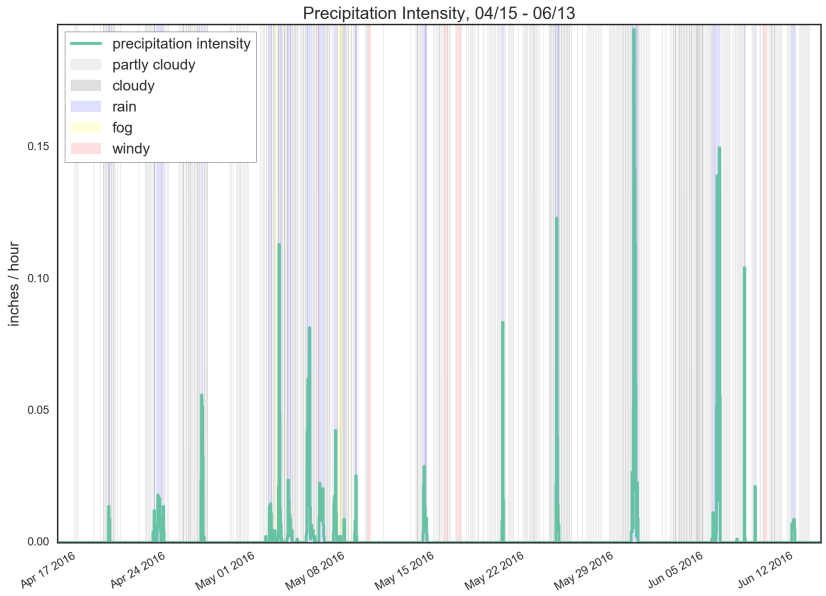


Figure 88: Precipitation Intensity during Test Period

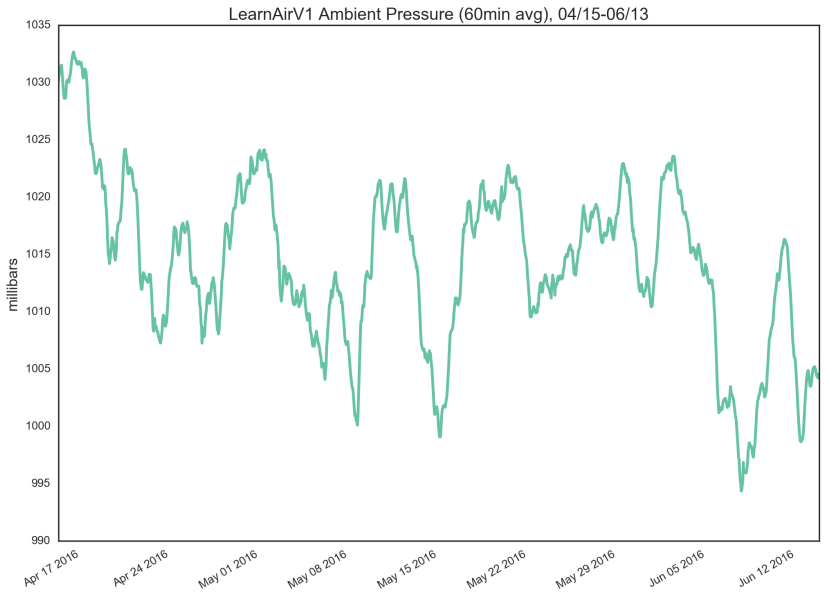


Figure 89: Ambient Pressure during Test Period

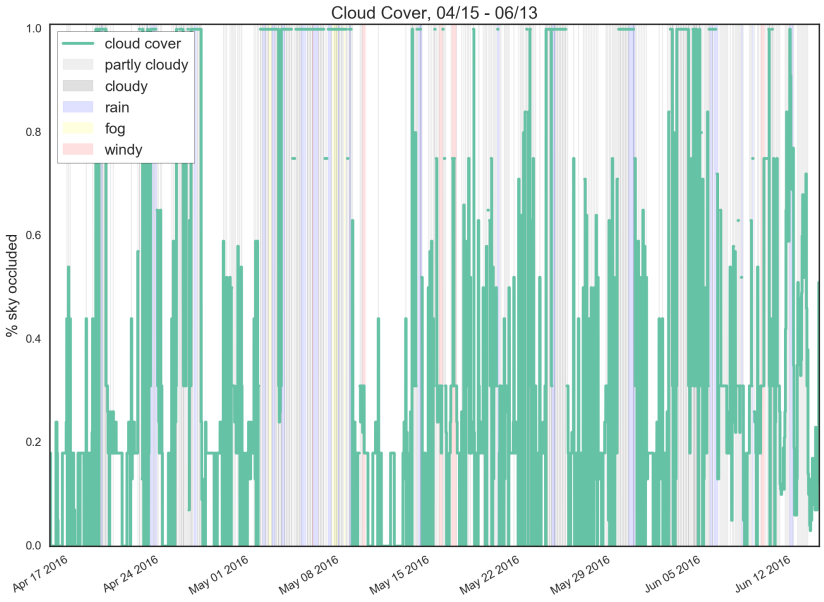


Figure 90: Cloud Cover during Test Period

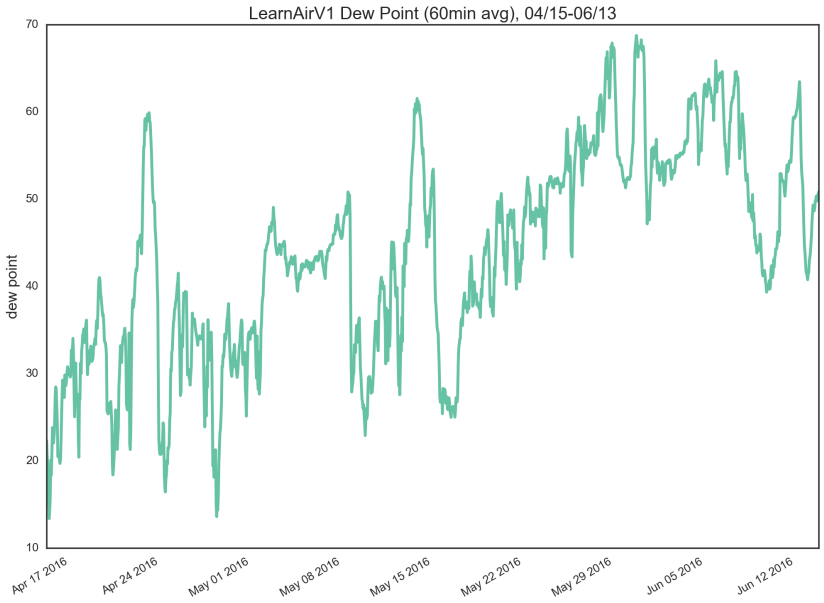


Figure 91: Dew during Test Period

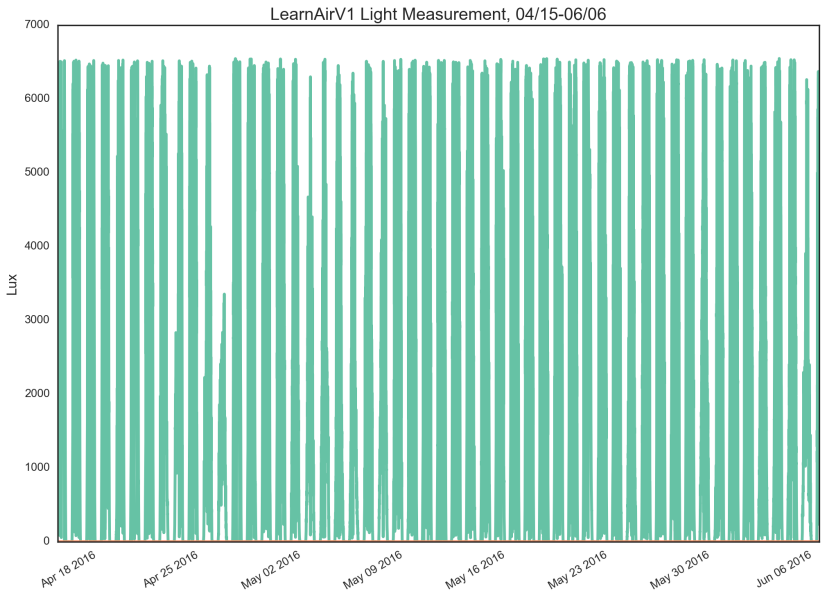


Figure 92: Lux during Test Period

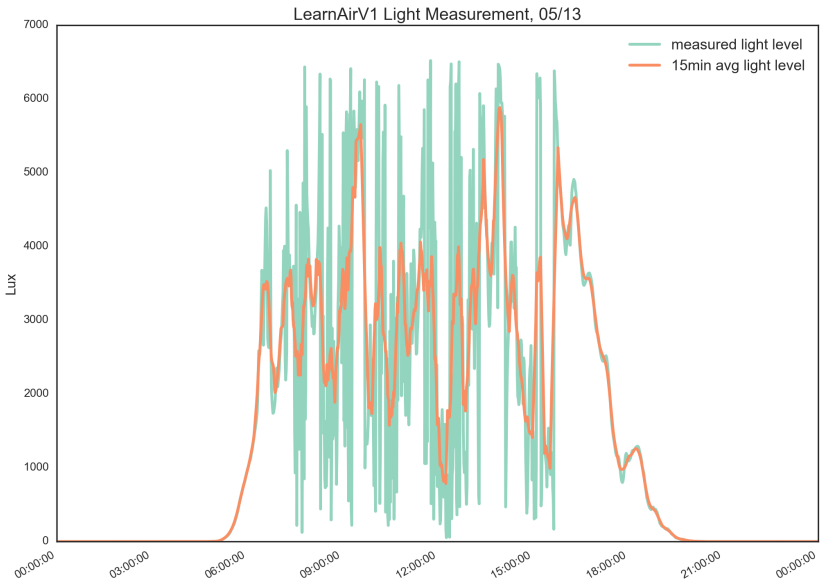


Figure 93: Lux during Test Period

Data Pre-Processing

Features

SmartCitizen CO

SmartCitizen NO2

Sharp Dust Sensor

AlphaSense CO

AlphaSense NO2

AlphaSense O3

Bibliography

[Tseng, 2016] Tseng, T. (2016). Spin: Examining the role of engagement, integration, and modularity in supporting youth creating documentation. In *Proceedings of the 2016 conference on Designing Interactive Systems*. ACM.