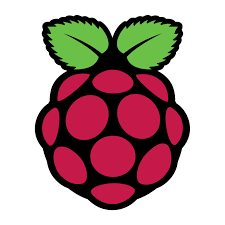
IoT with AutoML for Manufacturing

By: Andrew Jones

Contents

[Introduction & Background 2](#_Toc518030614)

[Process and Methods 2](#_Toc518030615)

[IoT Device Configuration 2](#_Toc518030616)

[AWS Python SDK (Boto3) and Hosting Solution 3](#_Toc518030617)

[AutoML with TPOT 5](#_Toc518030618)

[AutoML Implementation 6](#_Toc518030619)

[Benefits & Results 8](#_Toc518030620)

[Practical Concerns & Cost 9](#_Toc518030621)

[Conclusion 10](#_Toc518030622)

[Project Experience and DS Program Preparation 10](#_Toc518030623)

[Sources 11](#_Toc518030624)

# Introduction & Background

One of the most common complaints for Data Science & AI in general is that time to implementation for any practical business application is long and arduous. Among others, this is often due to reasons such as the lack of valuable & correctly pre-processed data for modeling, the need for connectivity between the source of the data and the predictive model, and the lengthy steps of model selection, through hyperparameter tuning & validation. This is no different in the context of manufacturing, where a diverse array of machines, widgets, locations & often low fault tolerances result in the need for customized models that cannot be cost effectively developed through traditional means. **This project’s goal will therefore be showcasing how low cost IoT devices & sensors can be combined with cloud computing storage and AutoML packages to very quickly produce powerful machine learning algorithms for manufacturing at scale.** The goal of this project is not however, to produce the best classifier model possible on the manufacturing dataset which is used to showcase the final AutoML portion.

This project combines my interest in the three areas of machine learning, cloud computing and IoT devices, the last of which I had no prior experience with. Due to the time spent to develop the sensor device, it was always impractical to have a “live” implementation of it for optimization within a manufacturing plant. Therefore, I’ve used an open source manufacturing dataset (for semiconductor manufacturing defect detection) to showcase the AutoML piece. The IoT sensor device will then simply produce the same types of fields, based on the same signals (e.g. temperature) that can be found within the open source dataset.

# Process and Methods

This section will be composed of 4 different sub-sections: IoT Device Configuration, AWS Python SDK (Boto3) and Hosting Solution, AutoML with TPOT, and AutoML Implementation

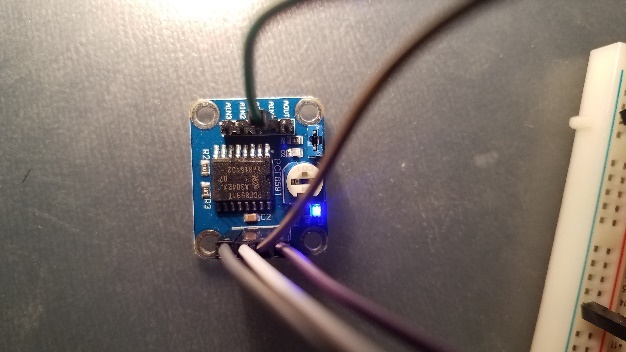
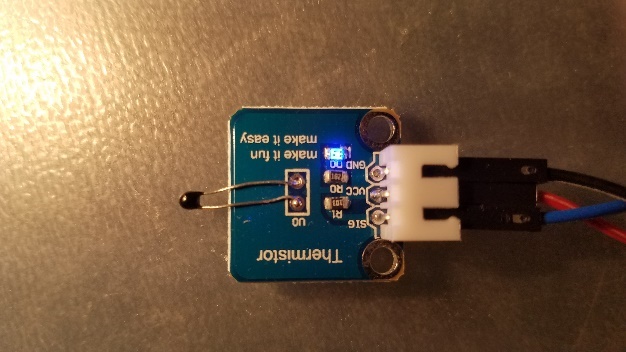
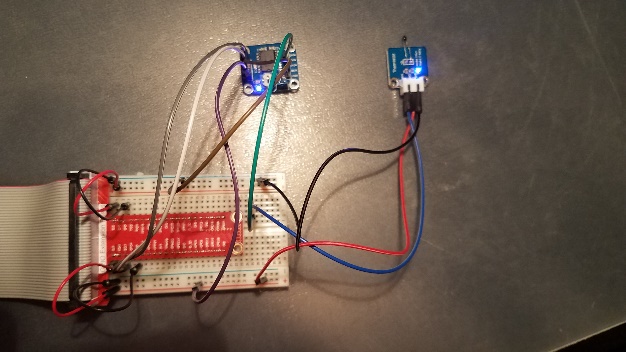
### IoT Device Configuration

Firstly, to address the problems around valuable & correctly pre-processed data, I’ve utilized an IoT solution which can ingest data from a variety of environmental factors including: temperature, sound, distance, magnetism, humidity the presence of an object or not, etc. These are precisely the types of signals used in a manufacturing setting to track the progression of units and their consistency in product design. In this case I’ve built the device that would detect these signals using a Raspberry Pi B+, which is a microcontroller device typically running a special Linux OS. This device combined with very cheap aftermarket sensors and a bit of electrical engineering, can detect many of the environmental factors mentioned above. Many of the baseline Python scripts and circuit arrangements needed to configure the sensors are available from the manufacturers, making it a very low barrier to entry in producing a sensor device.

The scripts used to publish the signals can be referenced in the following file. Please note this script needs to be executed in Python 2 from the Raspberry Pi to run: *Applied\_Research\_Project\_SensorMessage\_v2.3\_NoCredentials.py*

For this project I’ve bought the “Canakit Ultimate Starter Kit” package for the Raspberry Pi which can be found [here](https://www.canakit.com/raspberry-pi-3-ultimate-kit.html). This starter kit comes with the array of necessary components needed to startup the Raspberry Pi (power supply, HDMI cable, heat sinks, microSD card) as well as common items needed for beginner projects (GPIO[[1]](#footnote-1) Breakout board, LED’s, Cables). This microSD card comes pre-installed with the Raspbian OS[[2]](#footnote-2), meaning I don’t need to download & install that image to the SD card from another device first. However, this starter kit did not come with the sensors needed in our specific manufacturing context. So, I’ve purchased the SunFounder 37 Modules Sensor Kit v2.0 which can be found [here](https://www.sunfounder.com/starterkit/rpi/sensor-kit/rpi2-sensorv2.html). This kit includes a variety of small sensors that can be used with their pre-defined scripts & the GPIO pins to produce a digital record of an environmental signal, such as temperature, distance or humidity. The digital record of the signal is at first simply printed to the Python console. It is up to the user to do something with that message and compile them in a meaningful way for further analysis.

Below are the pictures of the Raspberry Pi configuration with the Thermistor sensor setup. This sensor is very accurately able to detect the ambient temperature when used in combination with the PCF8591[[3]](#footnote-3) module. Due to financial limitations and space limitations on the breadboard (with just one Raspberry Pi 3 B+), this will be the only signal I’m producing to showcase the functionality. However, the process to configure other Raspberry Pi’s and sensors as needed for a live setting would be exactly the same.



*Figure 1: From left to right, Raspberry Pi 3 B+, GPIO Breakout board with sensors, Thermistor Sensor Module, PCF8591 Module*

### AWS Python SDK (Boto3) and Hosting Solution

Secondly in addressing the problem of connectivity between the data source & the predictive model, I’ve created an instance of AWS Dynamo DB (No SQL database) which I will connect to programmatically utilizing the Boto3[[4]](#footnote-4) Python package & API. The Boto3 package is a package you install with Python (like any other) and which is then configured to pass credentials and directly access AWS services programmatically through their API. The Dynamo DB instance is a very cheap, NoSQL database where I will be publishing & storing the sensor data. In practice, this would become the data source that the AutoML model is trained on (either in AWS for scaling CPU’s / GPU’s or locally for cost savings) and then the same data source could be used for the implementation and real-time classification or regression scenario. DynamoDB is a great solution for easy scalability so that the read / write frequency can adjust depending on the latency requirements of the scenario.

While the connectivity could have just as easily been configured simply across my Wi-Fi network to a local database, the decision to construct the architecture in the cloud makes sense for a variety of reasons. First, are all the usual benefits of cloud architecture such as scalability & managed infrastructure. At the speed and scale manufacturing data is often aggregated with thousands of simultaneous units receiving hundreds of measurements; it can be a dizzying pace of records & size of data that requires you to have scalable architecture to cope. Having the management of servers outsourced brings the usual cost savings with labor to upgrade patches, provision storage as needed, and configuration. Second, and more specific to this use case, the implementation time to configure the AWS service over physical servers would be much less. The only physical hardware you need is the Raspberry Pi and sensors themselves. Lastly, AWS offers a service called “Greengrass” which would allow you to do the computation & AutoML “at the edge”. In certain scenarios where you need extremely low latency, or there is a problem with internet connectivity it would be beneficial to have the model implementation (if not the training as well) on the physical sensor devices. However, for this exercise due to cost concerns, I did not pursue that style of offering. Below are some snippets relating to the configuration of DynamoDB…..

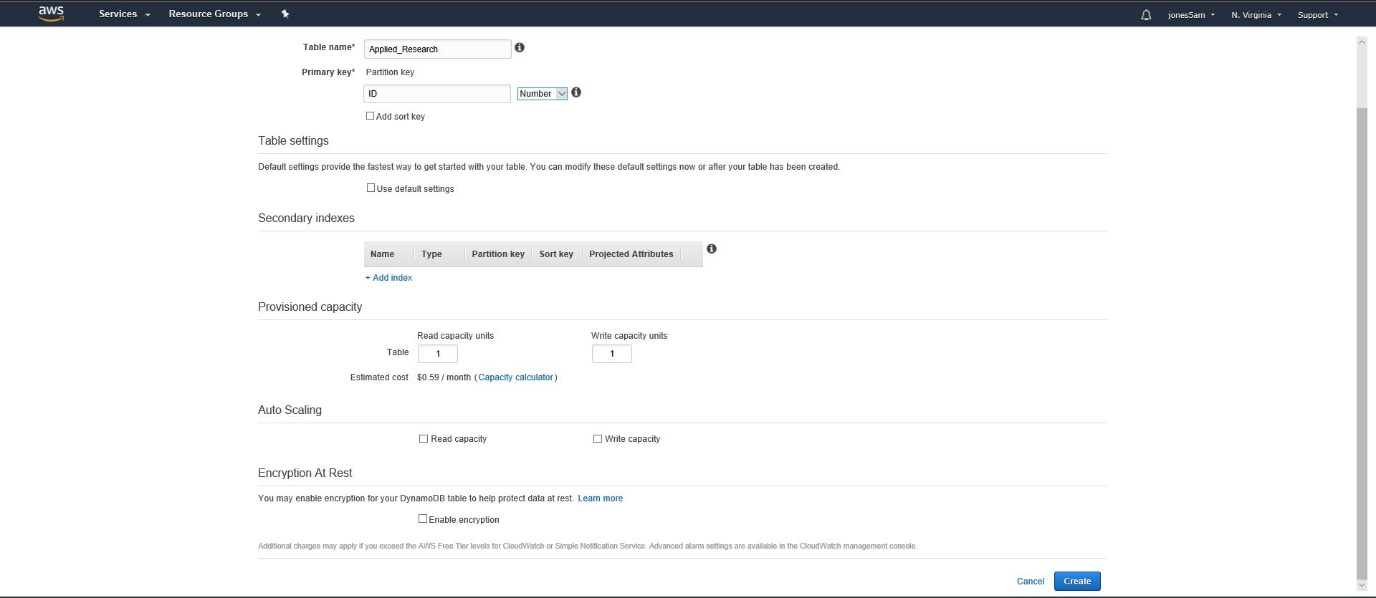


Figure 2: Configuring the DynamoDB Table

…. And the inserting of records



Figure 3: Code snippet for inserting a temperature record

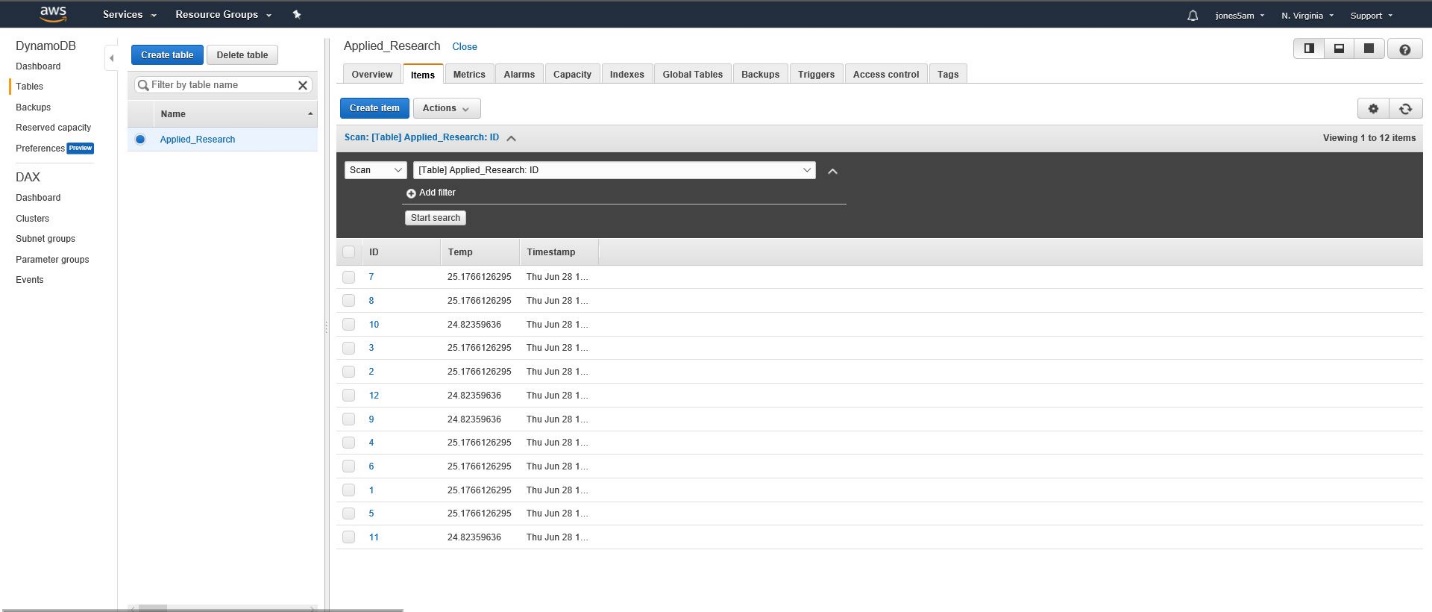


Figure 4: Temperature records correctly inserted into the DynamoDB table

### AutoML with TPOT

Thirdly, to address the problems around model selection, there are an array of packages & tools pitched as “AutoML” that employ different methods for automated model selection & hyperparameter tuning. Some of these tools are embedded within other more traditional analytics application (like RapidMiner), some are newer tools sold specifically to this purpose (like DataRobot), and finally some are open sources packages (TPOT, AutoSKLearn) which can be implemented for free. For this project I’ve made use of the open source TPOT package for feature engineering, model selection & hyperparameter tuning.

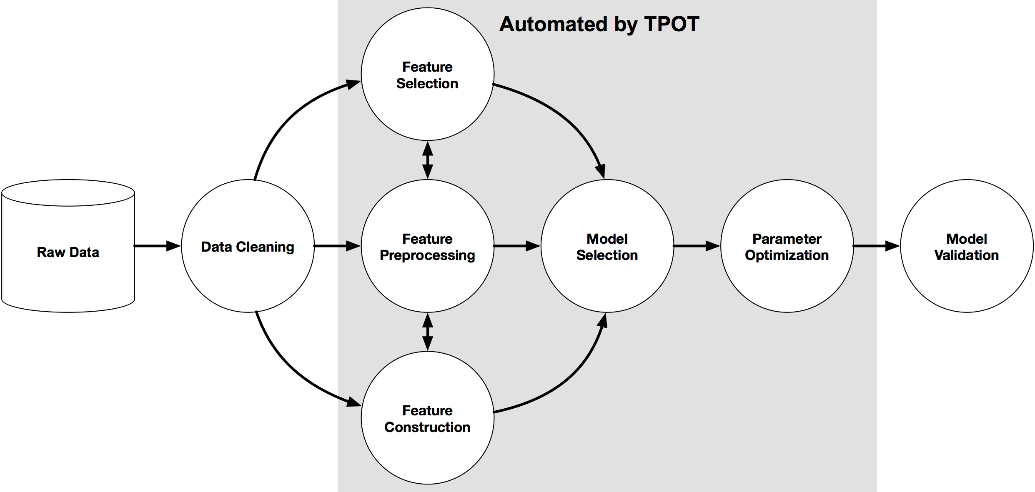


Figure 5: TPOT AutoML Progression

I chose this package over others available primarily because it had a high degree of documentation, it was built as a “wrapper” for SKLearn (and thereby NumPy, Pandas) and it could be implemented on a Windows environment natively. The other leading open source package for AutoML called Auto-SKLearn was also built on SKLearn, however it required either a Linux OS or virtualization thereof. While they employ different methods for finding the best solution, and Auto-SKLearn has won some impressive awards recently (Feurer, Klein, & Hutter, 2016), for the purposes of this demonstration I did not want to add another layer of configuration complexity with the Linux OS virtualization on my Windows computer. Due to cost savings & CPU speed / hard drive space I also didn’t want to run this hefty optimization on the AWS Cloud or Raspberry Pi respectively. Therefore, my Windows laptop (HP EliteBook) would have to suffice for this demonstration. However, in a real-world circumstance you would likely want to run the optimization in the cloud for performance reasons and could implement the model “at the edge” on the Raspberry Pi for reasons discussed already.

In terms of the TPOT package itself, like mentioned previously it is marketed as your “Data Science Assistant” and they are clear to indicate it is not a replacement tool. Obviously, there is a fair amount of work that goes in to data extraction, connectivity & manipulation to just get prepared for machine learning. The software works by using genetic programming to “search over a broad range of feature constructors, feature selectors, models, and parameters to find a series of operators that minimize the error of the model predictions.” By having a “score” output parameter for each algorithm, they essentially have turned model selection, some parts of feature engineering & hyperparameter tuning into a massive grid search problem. In my instance (with 1436 records X 420 fields = 603,120 values) it in all took about 3 hours to converge on a solution. In this case, I’ve invoked the TPOTClassifier (as opposed to TPOTRegressor) package which is enabled with the following classification algorithms:

|  |  |  |
| --- | --- | --- |
| * sklearn.naive\_bayes.GaussianNB | * sklearn.ensemble.ExtraTreesClassifier | * sklearn.svm.LinearSVC |
| * sklearn.naive\_bayes.BernoulliNB | * sklearn.ensemble.RandomForestClassifier | * sklearn.linear\_model.LogisticRegression |
| * sklearn.naive\_bayes.MultinomialNB | * sklearn.ensemble.GradientBoostingClassifier | * xgboost.XGBClassifier |
| * sklearn.tree.DecisionTreeClassifier | * sklearn.neighbors.KNeighborsClassifier |  |

Figure 6: TPOT Classification Algorithms

### AutoML Implementation

For this demonstration of AutoML within a manufacturing context I’ve used an open source dataset from the UCI Machine Learning Repository called SECOM which can be found [here](http://archive.ics.uci.edu/ml/machine-learning-databases/secom/secom.names). This dataset describes a modern complex semiconductor manufacturing process with originally 591 unlabeled features and 1567 records that may or may not have predictive power towards the identification of labelled defects. In practice I needed to eliminate some features & records however due to NaN and only zero as values features, so my final tally is only 1436 records and 420 features. I believe that in a manufacturing context, eliminating the features that are unreliable, only have zeros, and then records that still have any NaN values horizontally is a logical data cleaning progression for three reasons. First, the features that are unreliably accumulated (>25 missing records) should not be used as they cannot be depended on to even exist for a given unit. The 25 missing records threshold was derived simply because of a large gap inherent in the number of missing records for any feature. The lowest missing number of records for any feature above 25 missing records was 200+. Second, features that only have zero as their value for every record will obviously not provide any predictive power between classes. Third, after eliminating the frequently NaN features and features with zero’s as values, it makes sense to eliminate the records out of the remaining features that may still have NaN values. This is because just the presence of a NaN value means that likely something is wrong with the typical manufacturing process and we don’t want to include those outliers. All in all, this final step eliminated an additional 131 records.

Now with the cleaned dataset, I conducted some Exploratory Data Analysis. In this dataset we have only 100 defective units to 1336 normal units leaving us with a clearly unbalanced dataset. Again, with respect to the minimalist approach of implementing machine learning, I wanted to see if TPOT could correctly handle this very common problem. It will likely lead to tree-based methods, which are best at handling this structure of dataset (How to Handle Unbalanced Classes in Machine Learning, 2017).

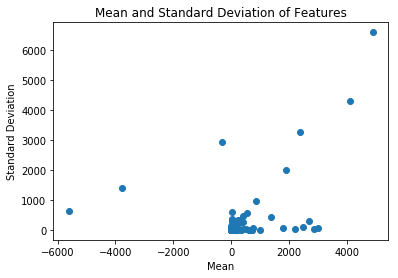
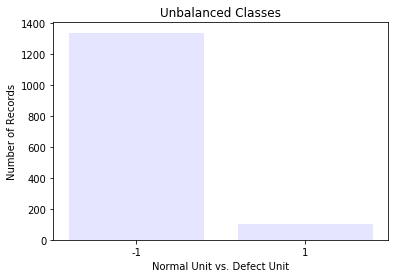


Figure 7: Unbalanced Classes Present Figure 8: Varied Values within Unlabeled Features

Now with the EDA completed, the final step for the data cleaning is the simple train\_test\_split function from sklearn. Now that the data is fully cleaned and prepared (the main part of this workflow that cannot be automated), we’re ready to invoke the TPOT classifier.



Figure 9: TPOT Code for Pipeline Creation

This section of code ran for about 3 hours testing all models, features and hyperparameters and ranking them based on ROC\_AUC. Unsurprisingly in the end, the TPOT algorithm decided on the Random Forest Algorithm owing to the unbalanced dataset. It produced a score of 77.47% AUC and the following “pipeline” which can be implemented to classify future records as they are produced.



Figure 10: TPOT Output Pipeline for Random Forest Classifier

# Benefits & Results

While there were some compromises along the way that were made for the sake of cost or convenience, I believe the results of the project are certainly positive. The goal was not to make the “best” classifier possible on this particular dataset – that has undoubtedly already been done (Hung, n.d.). The goal of this project was proving that the whole workflow from data generation via sensors, through cloud computing for connectivity / storage, to AutoML for machine learning classifiers is an efficient and effective solution to the manufacturing domain. It’s truly the entire “workflow” from an issue existing, to gathering data, creating architecture to store the data, and running ML algorithms to optimize the situation. Many organizations spend millions on even just one piece of that puzzle, and my proposition here is that you don’t need to. Effective cheap, customizable, and/or open source tools are available to work in concert for these problems. While the barriers to configuring them have also been vastly lowered so that they no longer require deep domain experts such as:

* Electrical Engineers to design sensors capable of ingesting relevant signals
* Cloud Computing Architects for connecting and storing the data as it’s produced
* Data Scientists and Machine Learning experts for creating custom ML models

It is also worth mentioning that these same concepts could certainly apply to more domains than just manufacturing. The manufacturing industry was chosen simply because of its relevance to my current work, but many industries are being disrupted by the combination or individual components of IoT devices, Cloud Computing, and AutoML. Healthcare in particular stands out as potentially benefiting from all three.

# Practical Concerns & Cost

However as always, there are some practical concerns that should be mentioned about the project, which were quite simply not solvable within the time period or resources at my disposal. However, for a business hoping to provide offerings such as this, I imagine they would be minor and solvable concerns. These practical concerns include:

* Structure of Sensor Device – the Raspberry Pi is hugely customizable with many different cases and attachable components. For the purposes of this project, all the necessary components were not “boxed” into a single device that could be easily installed on a manufacturing machine due to the additional money it would require to mount them as such.
* Sensor Thresholds – the sensors I’ve acquired are not rated for extreme environments. In certain extreme environments (think molten iron for steel manufacturing) they would malfunction or be destroyed.
* Imagery Sensor Types – some manufacturing products or components may need to be monitored by camera sensors to identify anomalies. This could be done through computer vision. The TPOT package however does not have the Deep Learning algorithms necessary for computer vision (like CCN or YOLO). However, I hope to continue this project within my work and include computer vision (via a Raspberry Pi camera) as one of the features.

In addition, there are some additional costs that most businesses would need to incur as I made use of the AWS free tier. The listing of all costs includes:

* Raspberry Pi – $35.00
* Power Supply - $8.00
* Breadboard - $2.25
* Ribbon Cables - $1.50
* Sensors - $6.00 (average)
* AWS IoT - $0.042 per device per year
* AWS Dynamo DB
  + $0.00065 per write capacity unit (WCU), 1 WCU provides up to 3,600 writes per hour
  + $0.00013 per read capacity unit (RCU), 1 RCU provides up to 7,200 reads per hour

So, to put this pricing into the same situation that we needed 420 features and 1436 records to make an effective ML model and each feature was ingested from a Raspberry Pi; The total pricing would be:

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Number of Units | Estimated Cost Per Item | Total Cost |
| Raspberry Pi | 420 | $35.00 | $14,700 |
| Power Supply | 420 | $8.00 | $3,360 |
| Breadboard | 420 | $2.25 | $945 |
| Ribbon Cables | 840 (assume 4 each in each package) | $1.50 | $1,260 |
| Sensor | 420 | $6.00 | $2,520 |
| AWS Dynamo DB | 1437 WCU  1 RCU | $0.00065/ WCU  $0.00013 / RCU | $0.93  $0.00013 |
| TOTAL |  |  | $22,785.94 |

Figure 11: Comparatively Low Cost Solution

This pricing makes a couple of assumptions. The first assumption is that it took a year to accumulate the 1436 records which is probably overstated. It also assumes you need a separate Raspberry Pi for each feature, which is certainly not true since you could get GPIO pin expansion boards and configure multiple nearby sensors on one Raspberry Pi. This estimate doesn’t take into account the AWS Lambda or “offline” hardware that you would need to run the TPOT optimization. This estimate obviously also doesn’t consider the implementation time and labor. But this was not meant to be a comprehensive estimate. It is simply meant to show that with the right technologies this type of optimization can be done effectively and much more cheaply. In semiconductor manufacturing where each unit is relatively valuable, and the downstream costs of a faulty chip proceeding are high, this would be a huge bargain.

# Conclusion

So, in conclusion I believe these technologies can be combined to implement an effective classification or regression based solution for manufacturing and other domains. The cost of the solution is low, the configuration barriers have been significantly reduced with recent technologies / open source solutions, and the implementation time is only prolonged by the number of sensors you need to install. My hope is that by showcasing this combination of technologies, it can lead to a “live” implementation scenario in a realistic setting. Whether that be manufacturing or not.

*I hope this project was an interesting read and the value of the solution shows its merit. As always, please feel free to reach out with any advice, questions or concerns. Thanks!*

# Project Experience and DS Program Preparation

My experience with this project was certainly a great one. I knew from the onset that it would be interesting for a data science project if I could actually generate my own data. We’re often only involved after the point of the data being accumulated. Whether it’s a file off Kaggle, Data.Gov etc., this is always the starting point. This interest was combined with an initiative through my work to look at using AutoML for a manufacturing setting. So, I wanted to theorize an idea that combined those two interests and came upon the Raspberry Pi as a great device for utilizing sensors that can also be used in manufacturing. I had some familiarity with AWS from our Data Warehousing class as I used the AWS RDS service for constructing a MySQL database. I didn’t jump immediately to the conclusion that I should use AWS Boto3 & DynamoDB over sending the messages directly to my laptop. But once the theme of this really being a cost effective & simple solution came about, it was clear that I wouldn’t want users of this service to think they had to configure servers of their own. The AutoML piece through TPOT I thought was surprisingly easy to configure and run. The long optimization times that it takes, certainly do not warrant many trial & errors. So, I was happy that I only had to run the full optimization twice before working out all the details (mostly just the scoring method needed to be changed to ROC\_AUC from the simple default setting of accuracy, owing to the unbalanced classes)

I believe the Data Science program has prepared me for this project in several ways. As already mentioned I had some familiarity with AWS from the Data Warehousing project. So, the concept of Identity and Access Management (IAM), which was necessary to make the Raspberry Pi communicate with the AWS DynamoDB service, was familiar. For writing the various Python scripts that were needed to: read the sensor data, communicate that message to AWS DynamoDB, ingest that data, clean that data and run the TPOT AutoML functionality, we had the Data Mining course to build Python skills in. I would say the main part of the project which the Data Science program hadn’t prepared me for would be the electrical engineering portion, of designing the circuits necessary to power the Raspberry Pi sensors. This is obviously not an area that I would expect a Data Science program to tackle however, and there were many helpful guides readily available.

# Sources

1. AutoSKLearn wins ChaLearn AutoML challenge: (Feurer, Klein, & Hutter, 2016)
2. How to Handle Unbalanced Classes in Machine Learning: (How to Handle Unbalanced Classes in Machine Learning, 2017)
3. Data Analysis in Semiconductor Manufacturing: (Hung, n.d.)
4. TPOT Information: (What to expect from AutoML software, n.d.)
5. PCF8591 Information: <https://www.nxp.com/docs/en/data-sheet/PCF8591.pdf>
6. Setting up Boto3: <https://www.youtube.com/watch?v=VSpshSNY0vA&t=9s>
7. Configuring Thermistor Circuits: <https://www.sunfounder.com/learn/sensor-kit-v2-0-for-raspberry-pi-b-plus/lesson-18-temperature-sensor-sensor-kit-v2-0-for-b-plus.html>

# Scripts

1. **Applied\_Research\_Project\_pcf8591.py** - reference file for enabling the PCF8591 package
2. **Applied\_Research\_Project\_thermistor.py** – file for enabling the thermistor from Sunfounder sensor kit. Combined into next file which was ultimately used
3. **Applied\_Research\_Project\_SensorMessage\_v2.3\_NoCredentials.py** – file for temperature sensor readings & transmission to AWS Dynamo DB
4. **Applied\_Research\_Project\_TPOTprep\_v1.4.py –** file used to prepare, perform EDA and invoke the TPOT classifier on the SECOM (semiconductor) manufacturing dataset
5. **Applied\_Research\_Project\_tpot\_secom\_pipeline\_v2.0.py –** the output of the TPOTprep script giving us the highest scoring algorithm & tuned hyperparameters in the form of a “pipeline”

1. GPIO – General Purpose Input Output: set of programmable pins, that does not have a specific function. Programmable through packages like RPI.GPIO that come installed natively with the Raspbian version of Linux OS. [↑](#footnote-ref-1)
2. Raspbian OS - the Linux based recommended (especially for beginners) operating system made by the manufacturers for the Raspberry Pi. “Recommended” is said loosely however, as there are numerous operating systems available for a variety of purposes (e.g. Arcade Systems). [↑](#footnote-ref-2)
3. PCF8591 – single chip, single supply, low power 8 bit CMOS data acquisition device, analog to digital converter; functions of the device include analog input multiplexing, on-chip track and hold function, 8-bit analog to digital conversion and 8 bit digital to analog conversion. [↑](#footnote-ref-3)
4. Boto3 – “Boto is the Amazon Web Services (AWS) SDK for Python, which allows Python developers to write software that makes use of Amazon services like S3 and EC2. Boto provides an easy to use, object-oriented API as well as low-level direct service access.” [↑](#footnote-ref-4)