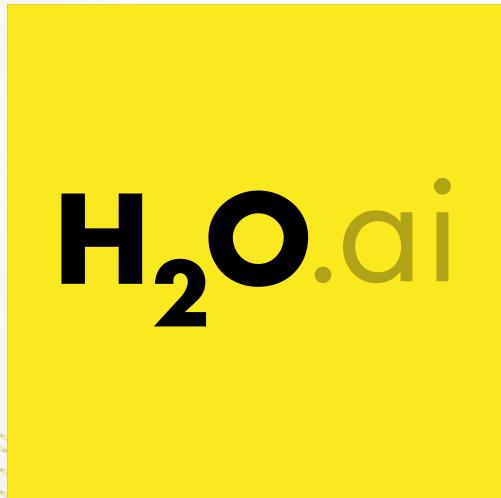


H2O Machine Learning and Kalman Filters for Machine Learning



Hank Roark
@hankroark
hank@h2o.ai

WHO AM I

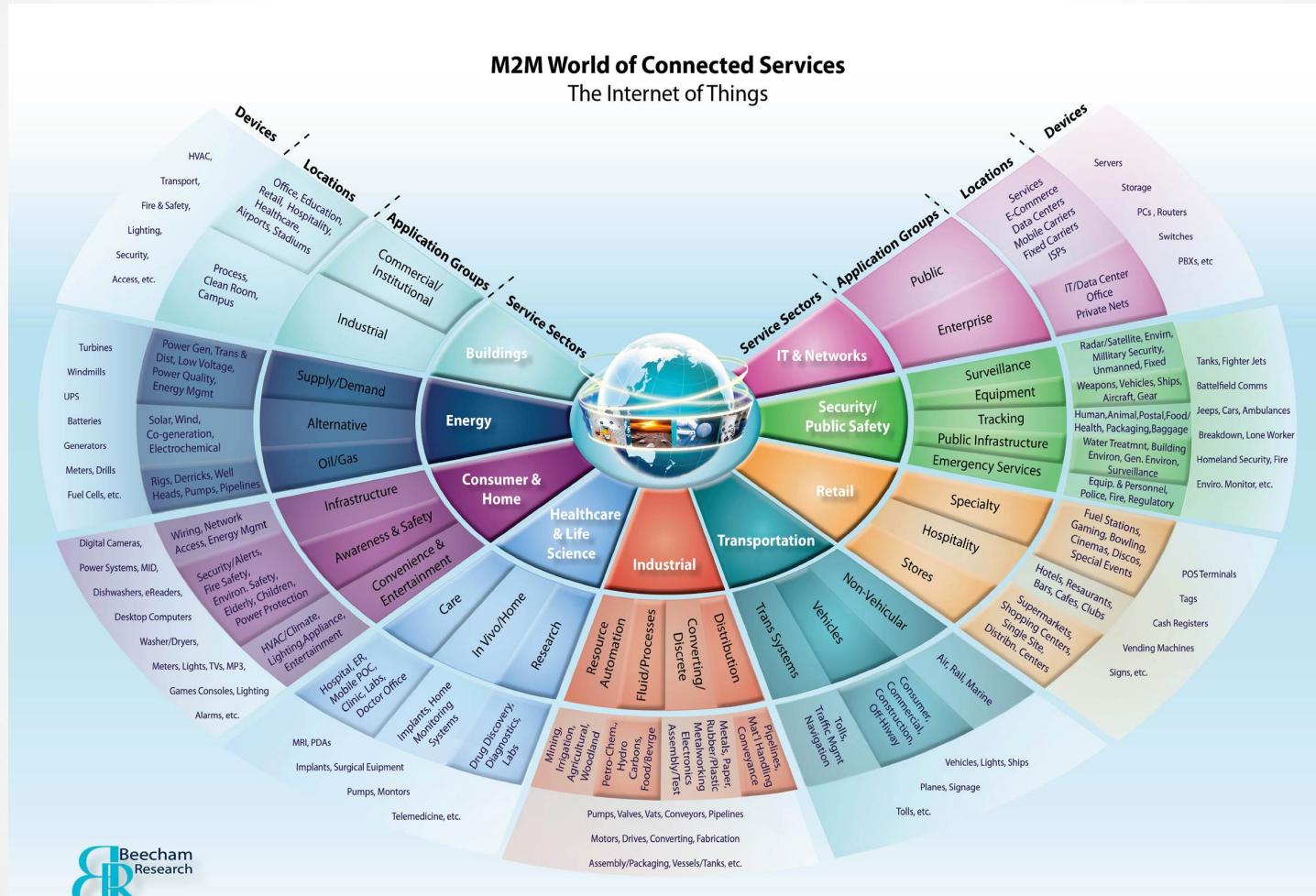
Lead, Customer Data Science @ H2O.ai

John Deere: Research, Software Product Development, High Tech Ventures
Lots of time dealing with data off of machines, equipment, satellites, weather, radar,
hand sampled, and on.
Geospatial, temporal / time series data almost all from sensors.
Previously at startups and consulting (Red Sky Interactive, Nuforia, NetExplorer, Perot
Systems, a few of my own)

Engineering & Management MIT
Physics Georgia Tech

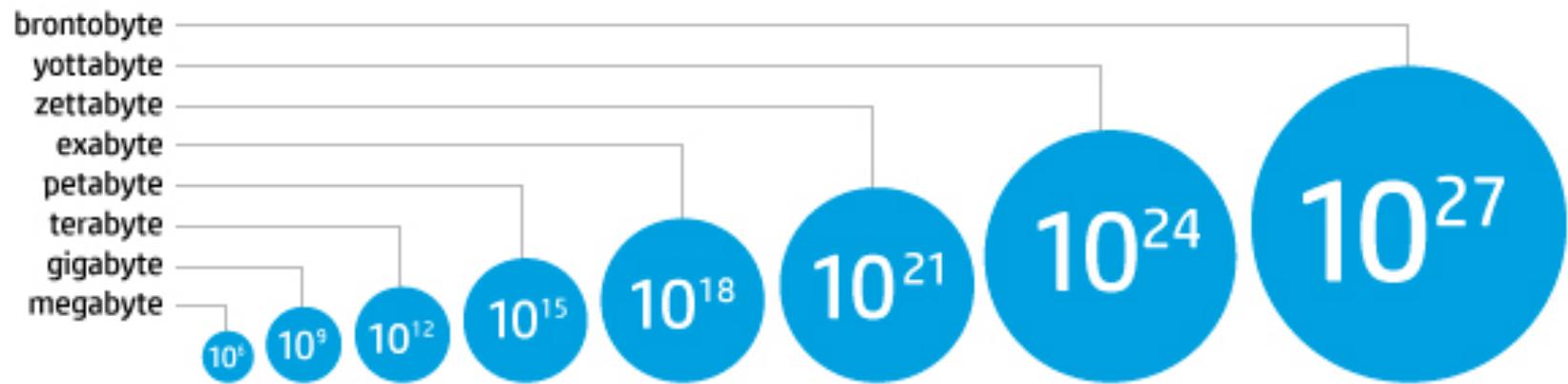
hank@h2oai.com
@hankroark
<https://www.linkedin.com/in/hankroark>

IF YOU ARE INTO DATA, THE IOT HAS IT



WHY THIS EXAMPLE?

Information & the Internet of Things



Today, data scientists max out at yottabytes, but soon, brontobytes will measure the volume of sensor data generated by the Internet of Things.

Source: HP

GET READY FOR BRONTOBYTES!!

WOW, HOW BIG IS A BRONTOBYTE?

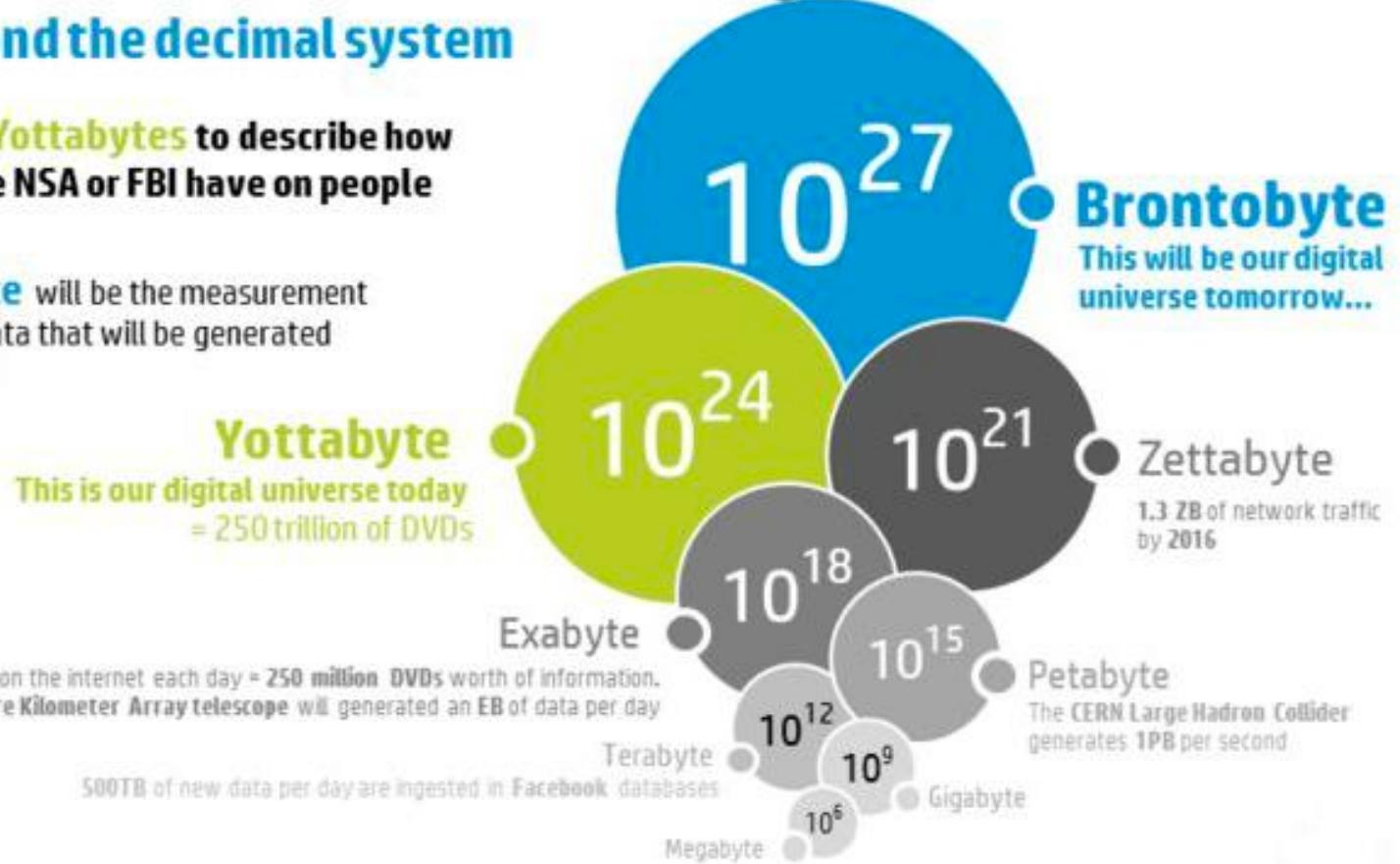
Information from the Internet of Things: We have gone beyond the decimal system

Today data scientist uses **Yottabytes** to describe how much government data the NSA or FBI have on people altogether.

In the near future, **Brontobyte** will be the measurement to describe the type of sensor data that will be generated from the IoT (Internet of Things)

1 EB of data is created on the internet each day = 250 million DVDs worth of information.
The proposed Square Kilometer Array telescope will generate an EB of data per day

500TB of new data per day are ingested in Facebook databases



This much data will require a fast OODA loop

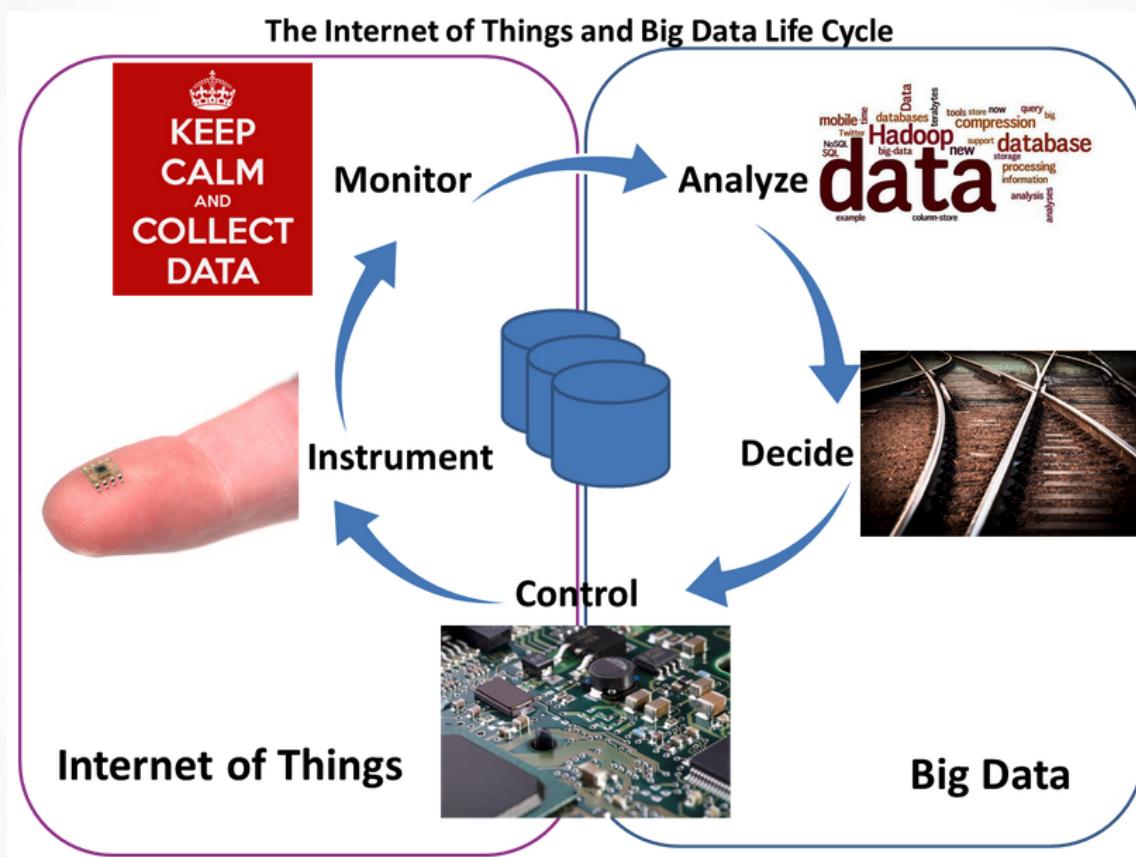


Image courtesy <http://www.telecom-cloud.net/wp-content/uploads/2015/05/Screen-Shot-2015-05-27-at-3.51.47-PM.png>

EXAMPLE FROM THE IOT

Domain: Prognostics and Health Management

Machine: Turbofan Jet Engines

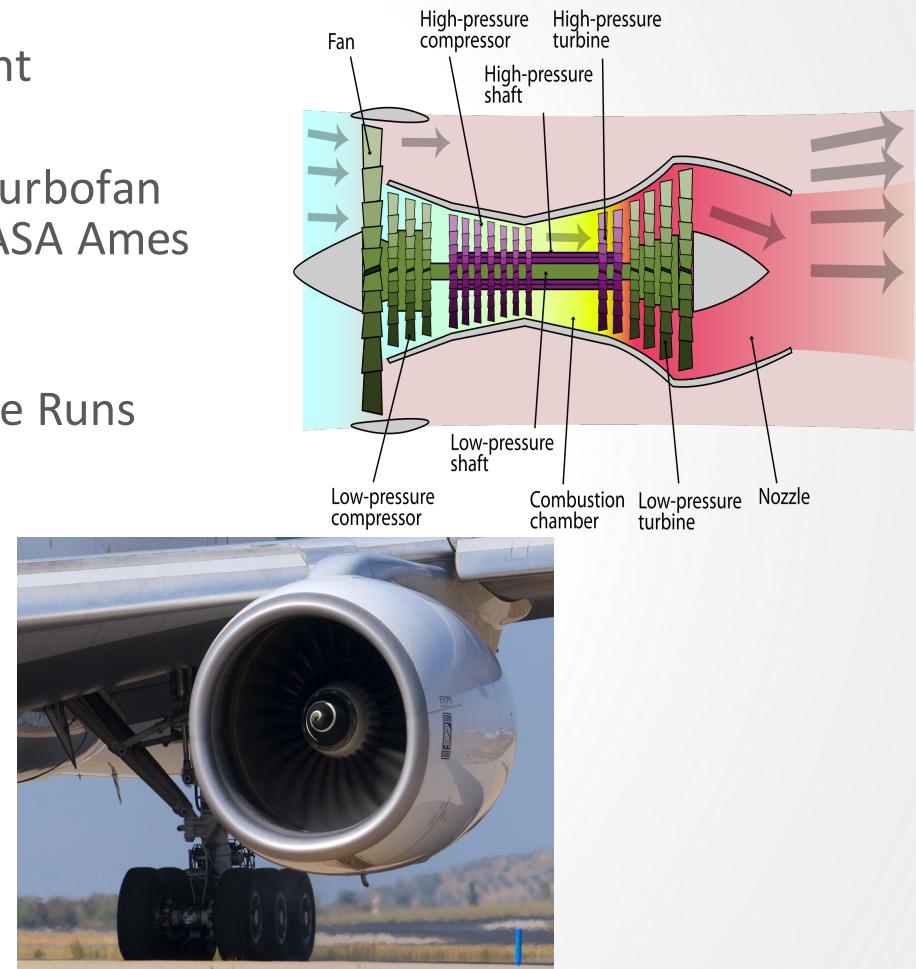
Data Set: A. Saxena and K. Goebel (2008). "Turbofan Engine Degradation Simulation Data Set", NASA Ames Prognostics Data Repository

Predict Remaining Useful Life from Partial Life Runs

Six operating modes, two failure modes,
manufacturing variability

Training: 249 jet engines run to failure

Test: 248 jet engines



LOADING DATA

```
train = h2o.upload_file("train_FD004.txt")
test  = h2o.upload_file("test_FD004.txt")
train.set_names(input_file_column_names);
test.set_names(input_file_column_names);
```

Parse Progress: [#####] 100%

Parse Progress: [#####] 100%

PYTHON (AND R) OBJECTS ARE PROXIES FOR BIG DATA

STEP 1



```
→ h2o_df = h2o.import_file("hdfs://path/to/data.csv")
```

Python user

PYTHON (AND R) OBJECTS ARE PROXIES FOR BIG DATA

STEP 2

Python



2.1

Python function call

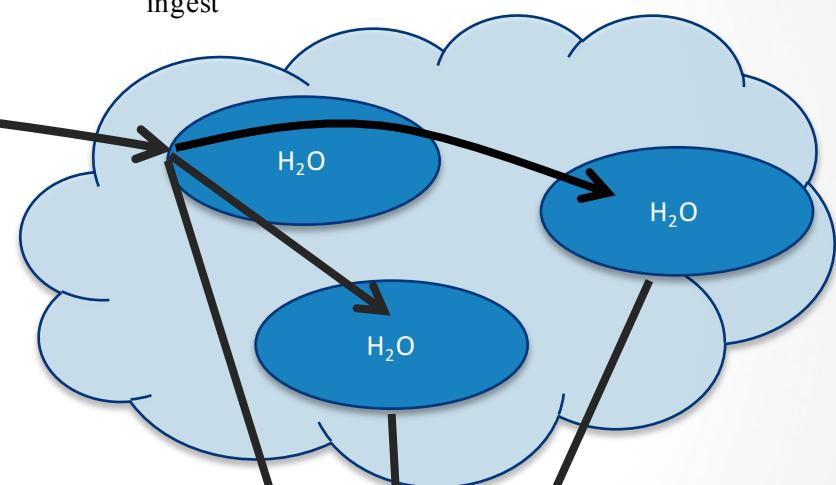
2.2

HTTP REST API
request to H₂O
has HDFS path

2.3

Initiate distributed
ingest

H₂O Cluster



2.4

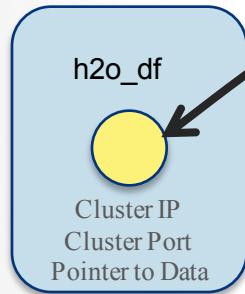
Request data
from HDFS

HDFS
NFS
S3

PYTHON (AND R) OBJECTS ARE PROXIES FOR BIG DATA

STEP 3

Python



3.4

h2o_df object created
in Python

3.3

Return pointer to data
in REST API JSON
Response

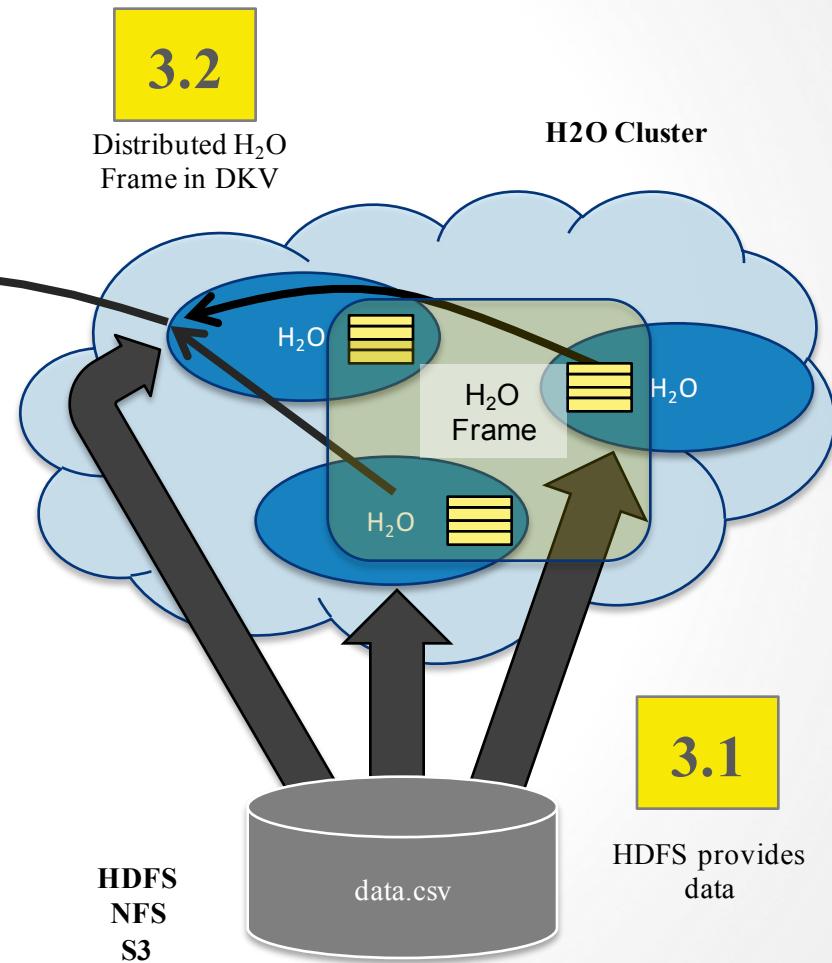
3.2

Distributed H₂O
Frame in DKV

H₂O Cluster

3.1

HDFS
provides
data



SUMMARY STATISTICS

```
train.describe()
```

Rows: 61,249 Cols: 26

	UnitNumber	Cycle	OpSet1	OpSet2	OpSet3	SensorMeasure1	SensorMe
type	int	int	real	real	int	real	real
mins	1.0	1.0	0.0	0.0	60.0	445.0	535.48
mean	124.325180819	134.311417329	23.9998233424	0.571346890561	94.0315760257	472.882435468	579.42005
maxs	249.0	543.0	42.008	0.842	100.0	518.67	644.42
sigma	71.9953498537	89.7833894132	14.7807216523	0.310703444054	14.2519539188	26.4368316429	37.342646
zeros	0	0	162	4776	0	0	0
missing	0	0	0	0	0	0	0
0	1.0	1.0	42.0049	0.84	100.0	445.0	549.68
1	1.0	2.0	20.002	0.7002	100.0	491.19	606.07
2	1.0	3.0	42.0038	0.8409	100.0	445.0	548.95
3	1.0	4.0	42.0	0.84	100.0	445.0	548.7
4	1.0	5.0	25.0063	0.6207	60.0	462.54	536.1
-	--	--	--	--	--	--	--

CREATE THE TARGET VARIABLE

```
def add_remaining_useful_life(h2o_frame):
    """
    Calculate total cycles for each unit
    grouped_by_unit = h2o_frame.groupby(by=[ "UnitNumber" ])
    max_cycle = grouped_by_unit.max(col="Cycle").frame

    # Merge the max cycle back into the original frame
    result_frame = h2o_frame.merge(max_cycle)

    # Calculate remaining useful life for each row
    remaining_useful_life = result_frame[ "max_Cycle" ] - \
        result_frame[ "Cycle" ]
    result_frame[ "RemainingUsefulLife" ] = remaining_useful_life

    # drop the un-needed column
    result_frame = result_frame.drop("max_Cycle")
    return result_frame

train_with_predictor = add_remaining_useful_life(train)
```

Calculate Total Cycles
For Each Unit

CREATE THE TARGET VARIABLE

```
def add_remaining_useful_life(h2o_frame):
    # Get the total number of cycles for each unit
    grouped_by_unit = h2o_frame.group_by(by=[ "UnitNumber" ])
    max_cycle = grouped_by_unit.max(col="Cycle").frame

    result_frame = h2o_frame.merge(max_cycle)

    # Calculate remaining useful life for each row
    remaining_useful_life = result_frame[ "max_Cycle" ] - \
        result_frame[ "Cycle" ]
    result_frame[ "RemainingUsefulLife" ] = remaining_useful_life

    # drop the un-needed column
    result_frame = result_frame.drop("max_Cycle")
    return result_frame

train_with_predictor = add_remaining_useful_life(train)
```

Append To
OriginalFrame

CREATE THE TARGET VARIABLE

```
def add_remaining_useful_life(h2o_frame):
    # Get the total number of cycles for each unit
    grouped_by_unit = h2o_frame.group_by(by=[ "UnitNumber" ])
    max_cycle = grouped_by_unit.max(col="Cycle").frame

    # Merge the max cycle back into the original frame
    result_frame = h2o_frame.merge(max_cycle)

    -----
    remaining_useful_life = result_frame[ "max_Cycle" ] - \
                           result_frame[ "Cycle" ]
    result_frame[ "RemainingUsefulLife" ] = remaining_useful_life

    # drop the un-needed column
    result_frame = result_frame.drop("max_Cycle")
    return result_frame

train_with_predictor = add_remaining_useful_life(train)
```

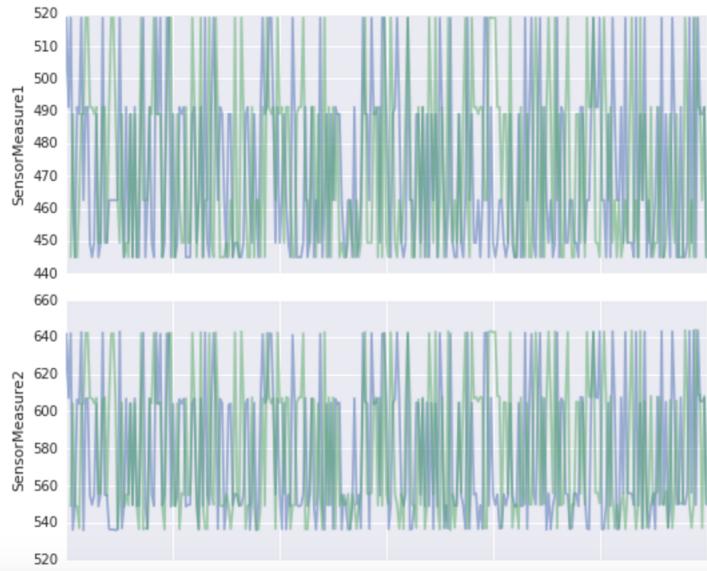
Create New
Feature of Cycles
Remaining

EXPLORATORY DATA ANALYSIS

```
sample_units = train_with_predictor["UnitNumber"] < 3
```

Boolean
Indexing

```
g = sns.PairGrid(data=train_pd,  
                  x_vars=dependent_var,  
                  y_vars=sensor_measure_columns_names + \  
                         operational_settings_columns_names,  
                  hue="UnitNumber", size=3, aspect=2.5)  
g = g.map(plt.plot, alpha=0.5)  
g = g.set(xlim=(300,0))  
g = g.add_legend()
```

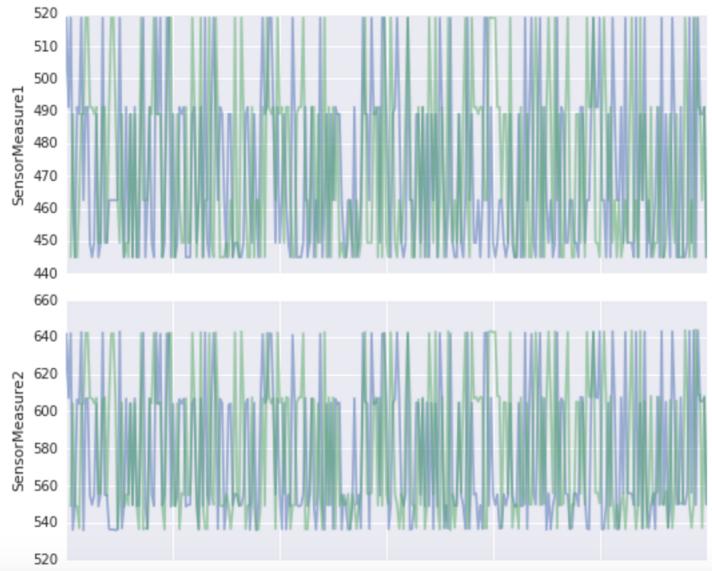


EXPLORATORY DATA ANALYSIS

```
train_pd = train_with_predictor[sample_units].as_data_frame()
```

Sample the
data to local
memory

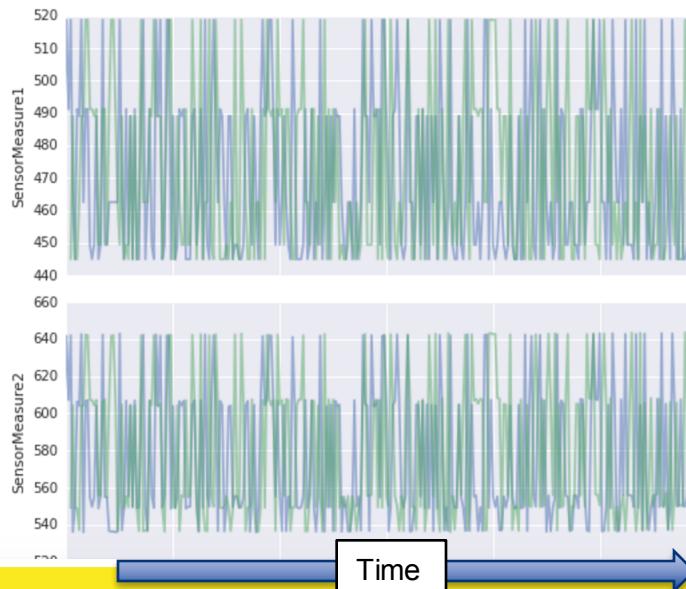
```
g = sns.PairGrid(data=train_pd,  
                  x_vars=dependent_var,  
                  y_vars=sensor_measure_columns_names + \  
                         operational_settings_columns_names,  
                  hue="UnitNumber", size=3, aspect=2.5)  
g = g.map(plt.plot, alpha=0.5)  
g = g.set(xlim=(300,0))  
g = g.add_legend()
```



EXPLORATORY DATA ANALYSIS

```
sample_units = train_with_predictor["UnitNumber"] < 3  
train_pd = train_with_predictor[sample_units].as_data_fra
```

```
g = sns.PairGrid(data=train_pd,  
                  x_vars=dependent_var,  
                  y_vars=sensor_measure_columns_names + \  
                         operational_settings_columns_names,  
                  hue="UnitNumber", size=3, aspect=2.5)  
g = g.map(plt.plot, alpha=0.5)  
g = g.set(xlim=(300,0))  
g = g.add_legend()
```



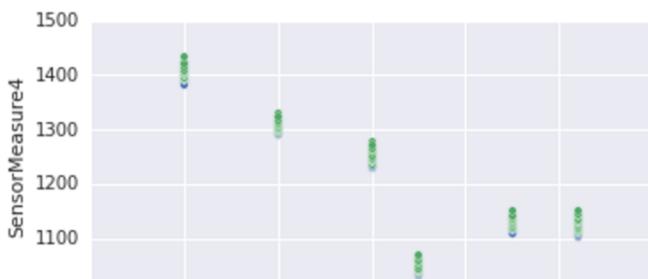
Use your favorite visualization tools

(Seaborn!)

Ugh,
where are
trends
over time

MODEL BASED DATA ENRICHMENT

```
g = sns.pairplot(data=train_pd,  
                  x_vars=[ "OpSet1", "OpSet2" ],  
                  y_vars=[ "SensorMeasure4", "SensorMeasure3",  
                           "SensorMeasure9", "SensorMeasure8",  
                           "SensorMeasure13", "SensorMeasure6" ],  
                  hue="UnitNumber", aspect=2)
```

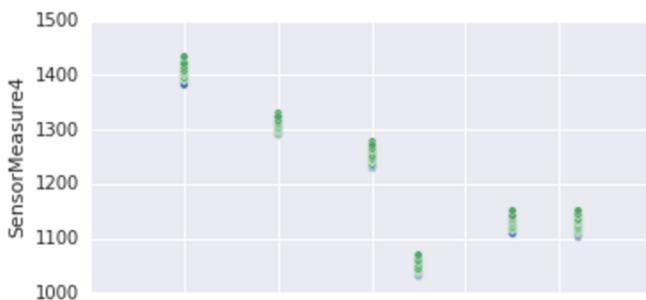


Sensor
measurements
appear in
clusters

Corresponding
to operating
mode!

FEATURE ENGINEERING

```
g = sns.pairplot(data=train_pd,  
                  x_vars=[ "OpSet1", "OpSet2"],  
                  y_vars=[ "SensorMeasure4", "SensorMeasure3",  
                           "SensorMeasure9", "SensorMeasure8",  
                           "SensorMeasure13", "SensorMeasure6"],  
                  hue="UnitNumber", aspect=2)
```



Use H2O k-means
to find cluster
centers

```
from h2o.estimators.kmeans import H2OKMeansEstimator
```

FEATURE ENGINEERING

```
: from h2o.estimators.kmeans import H2OKMeansEstimator  
  
: operating_mode_estimator = H2OKMeansEstimator(k=operating_  
operating_mode_estimator.train(x=operational_settings_colu  
training_frame=train_with_pr
```

Enrich existing data
with operating mode
membership

```
def append_operating_mode(h2o_frame, estimator):  
    operating_mode_labels = estimator.predict(h2o_frame)  
    operating_mode_labels.set_names(operating_mode_column_name);  
    operating_mode_labels = operating_mode_labels.asfactor()  
    h2o_frame_augmented = h2o_frame.cbind(operating_mode_labels)  
    return h2o_frame_augmented  
  
train_augmented = append_operating_mode(train_with_predictor,operating_mode_estimator)  
test_augmented = append_operating_mode(test,operating_mode_estimator)
```

MORE FEATURE ENGINEERING

```
def standardize_by_operating_mode(train, test):
    t = train.group_by(operating_mode_column_name).\
        mean(sensor_measure_columns_names).\
        sd(sensor_measure_columns_names).frame

    s = train.merge(t)
    r = test.merge(t)
    standardize_measures_columns_names = []
    for sensor_measure_column_name in sensor_measure_columns_names:
        include_this_measure = True
        # if any of the operating modes shows 0 or NaN standard deviation,
        # do not standardize that sensor measure,
        # nor use it in the model building
        for i in range(0,operating_modes):
            stdev = t[t["OperatingMode"] == str(i),"sdev_"+sensor_measure_column_name][0,0]
            if stdev == 0.0:
                include_this_measure = False
                break
        if include_this_measure:
            new_column_name = "stdized_"+sensor_measure_column_name
            standardize_measures_columns_names.append(new_column_name)
            s[new_column_name] = ((s[sensor_measure_column_name]-
                s["mean_"+sensor_measure_column_name])/
                s["sdev_"+sensor_measure_column_name])
            r[new_column_name] = ((r[sensor_measure_column_name]-
                r["mean_"+sensor_measure_column_name])/
                r["sdev_"+sensor_measure_column_name])
    return (s,r,standardize_measures_columns_names)

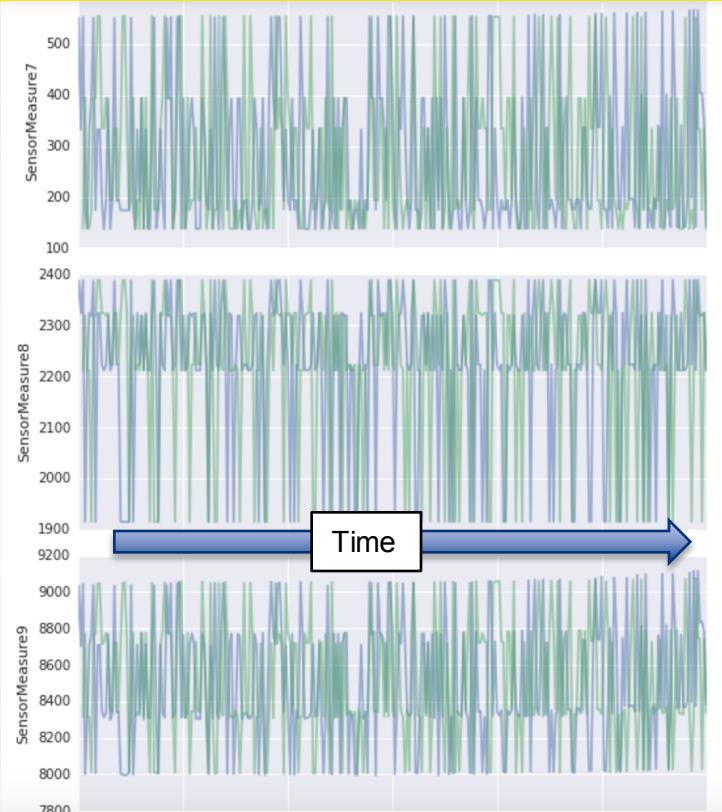
train_stdized, test_stdized, standardized_measures_columns_names = \
    standardize_by_operating_mode(train_augmented, test_augmented)
```

For non-constant
sensor
measurements
within an
operating mode,

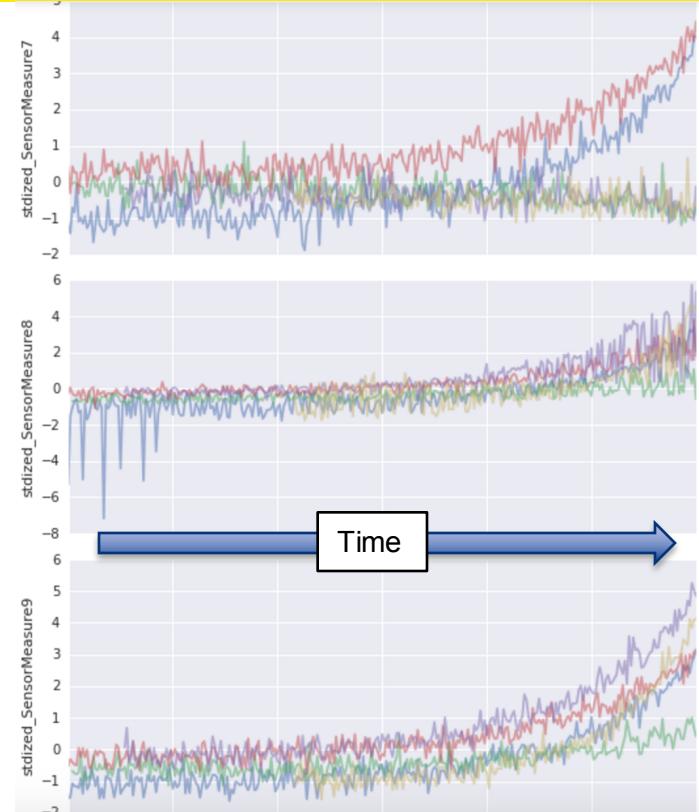
Standardize each
sensor measurement
by operating mode

Based on the
training data

TRENDS OVER TIME!



Before
H2O Data Preparation



Ready for
H2O Learning

MODELING - SIMPLE

```
from h2o.estimators.gbm import H2OGradientBoostingEstimator
```

```
gbm_regressor = H2OGradientBoostingEstimator(distribution="gaussian",
                                              score_each_iteration=True,
                                              stopping_metric="MSE",
                                              stopping_tolerance=0.001,
                                              stopping_rounds=5)
```

```
    .train(...  
          training_frame=train_final,  
          fold_column=fold_column_name)
```

Configure an
Estimator

MODELING - SIMPLE

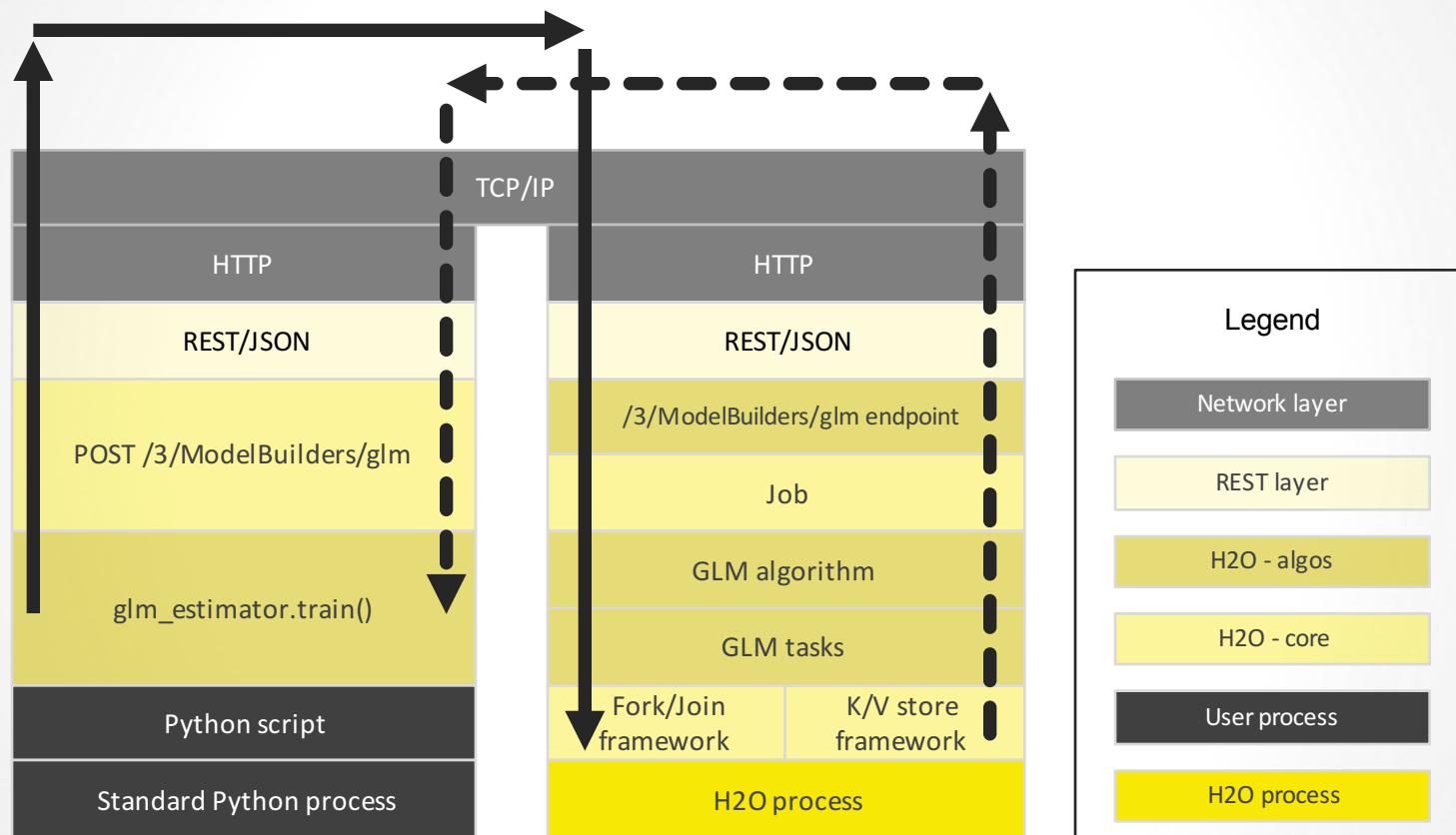
```
from h2o.estimators.gbm import H2OGradientBoostingEstimator

gbm_regressor = H2OGradientBoostingEstimator(distribution="gaussian",
                                              score_each_iteration=True,
                                              stopping_metric="MSE",
                                              stopping_tolerance=0.001,
                                              )

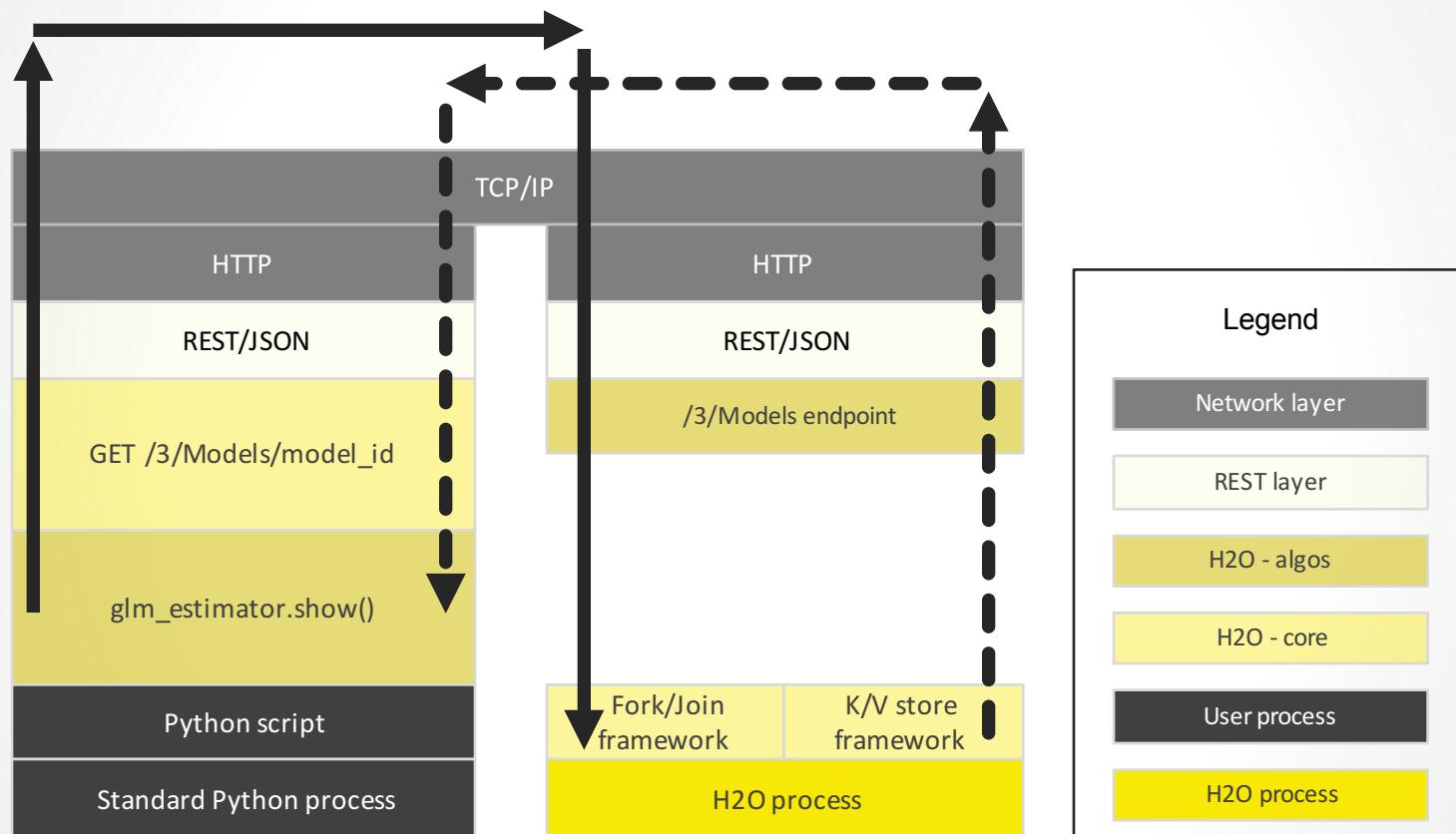
gbm_regressor.train(x=independent_vars, y=dependent_var,
                     training_frame=train_final,
                     fold_column=fold_column_name)
```

[Train an Estimator](#)

PYTHON SCRIPT STARTING H2O GLM



PYTHON SCRIPT STARTING H2O GLM



MODEL EVALUATION

```
: gbm_regressor
```

```
Model Details
```

```
=====
```

```
H2OGradientBoostingEstimator : Gradient Boos  
Model Key: GBM_model_python_1446901896856_
```

```
Model Summary:
```

number_of_trees	model_size_in_bytes	min_depth
40.0	17218.0	5.0

```
ModelMetricsRegression: gbm  
** Reported on train data. **  
  
MSE: 2163.66503487  
R^2: 0.731586024356  
Mean Residual Deviance: 2163.66503487  
  
ModelMetricsRegression: gbm  
** Reported on cross-validation data. **  
  
MSE: 2593.60830294  
R^2: 0.678249310944  
Mean Residual Deviance: 2593.60830294
```

Evaluate Performance
at a glance
in Python

MODEL EVALUATION

```
gbm_regressor
Model Details
=====
H2OGradientBoosting
Model Key: GBM_moc

Model Summary:


| number_of_trees | mo  |
|-----------------|-----|
| 40.0            | 172 |

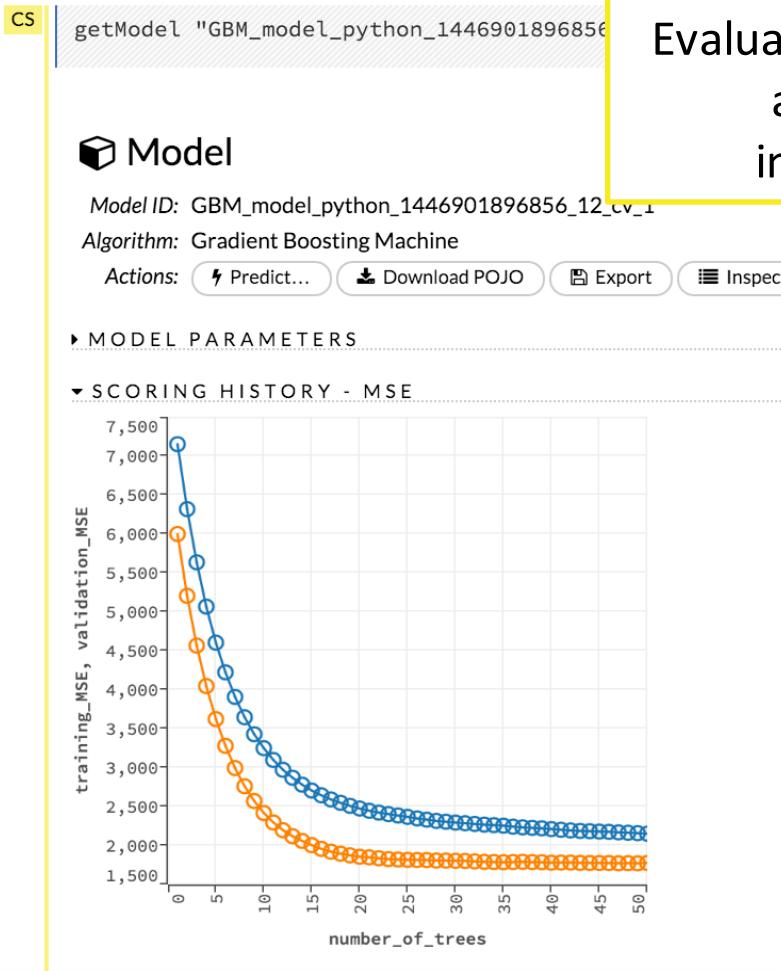


ModelMetricsRegression
** Reported on training set

MSE: 2163.66503487
R^2: 0.731586024356
Mean Residual Deviation: 42.055355

ModelMetricsRegression
** Reported on cross-validation set

MSE: 2593.60830294
R^2: 0.678249310944
Mean Residual Deviation: 49.855355
```



Evaluate Performance
at a glance
in H2O Flow

MODEL EVALUATION

```
: gbm_regressor
```

Model Details

=====

H2OGradientBoosting

Model Key: GBM_moc

Model Summary:

number_of_trees	mo
40.0	172

ModelMetricsRegression

** Reported on train

MSE: 2163.66503487

R^2: 0.731586024356

Mean Residual Devia

ModelMetricsRegression

** Reported on cross

MSE: 2593.60830294

R^2: 0.678249310944

Mean Residual Devia

CS

```
getModel "GBM_model_python_1446901896856_12_cv_1"
```



Model

Model ID: GBM_model_

Algorithm: Gradient Boo

Actions: Predict...

► MODEL PARAMETER

▼ SCORING HISTORY

ModelMetricsRegression

** Reported on train

MSE: 2163.66503487

R^2: 0.731586024356

Mean Residual Devia

ModelMetricsRegression

** Reported on cross

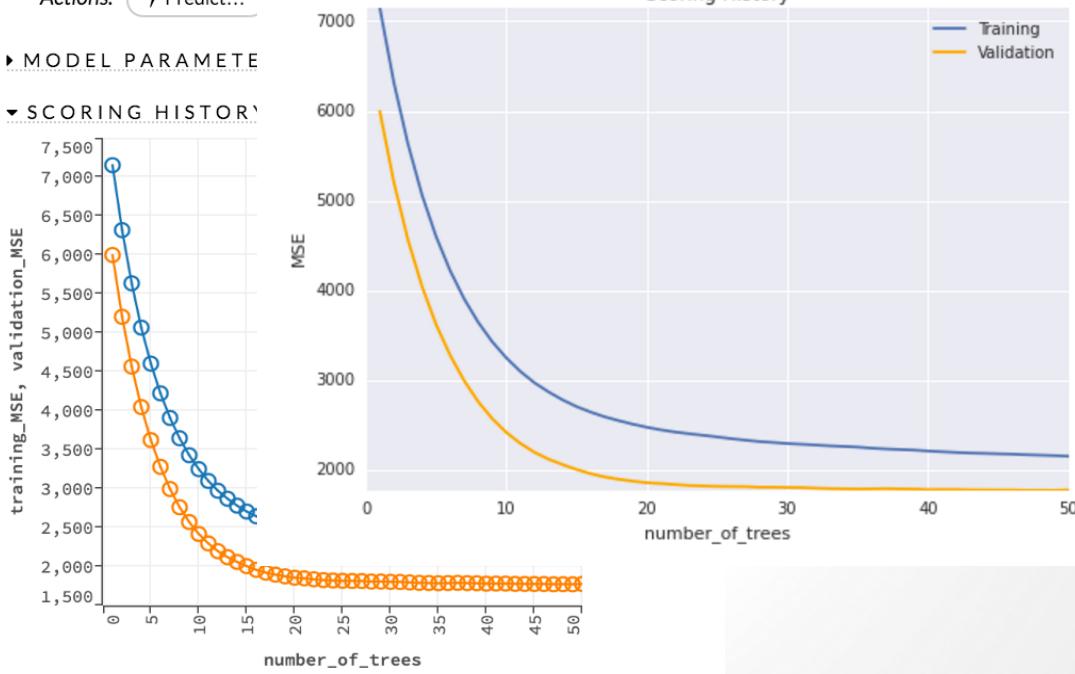
MSE: 2593.60830294

R^2: 0.678249310944

Mean Residual Devia

```
gbm_regressor.get_xval_models()[0].plot()
```

Scoring History



Evaluate Performance
at a glance
graphically in Python

CROSS VALIDATION

```
from h2o.grid.grid_search import H2OGridSearch

ntrees_opt = [1000]
max_depth_opt = [2, 5, 7]
learn_rate_opt = [0.01]
min_rows_opt = [5, 10, 15]
hyper_parameters = {"ntrees": ntrees_opt,
                     "max_depth": max_depth_opt,
                     "learn_rate": learn_rate_opt,
                     "min_rows": min_rows_opt}

gs = H2OGridSearch(gbm_regressor, hyper_params=hyper_parameters)

gs.train(x=independent_vars, y=dependent_var,
          training_frame=train_final,
          fold_column=fold_column_name)
```

Setup

Hyperparameter
Search Options

CROSS VALIDATION

```
from h2o.grid.grid_search import H2OGridSearch

ntrees_opt = [1000]
max_depth_opt = [2, 5, 7]
learn_rate_opt = [0.01]
min_rows_opt = [5, 10, 15]
hyper_parameters = {"ntrees": ntrees_opt,
                    "max_depth": max_depth_opt,
                    "learn_rate": learn_rate_opt,
                    "min_rows": min_rows_opt}

gs = H2OGridSearch(gbm_regressor, hyper_params=hyper_parameters)

gs.train(x=independent_vars, y=dependent_var,
          training_frame=train_final,
          fold_column=fold_column_name)
```

Configure
full full
grid search

CROSS VALIDATION

```
from h2o.grid.grid_search import H2OGridSearch

ntrees_opt = [1000]
max_depth_opt = [2, 5, 7]
learn_rate_opt = [0.01]
min_rows_opt = [5, 10, 15]
hyper_parameters = {"ntrees": ntrees_opt,
                    "max_depth": max_depth_opt,
                    "learn_rate": learn_rate_opt,
                    "min_rows": min_rows_opt}

gs = H2OGridSearch(gbm_regressor, hyper_params=hyper_parameters)

gs.train(x=independent_vars, y=dependent_var,
         training_frame=train_final,
         fold_column=fold_column_name)
```

Execute
grid search

CROSS VALIDATION

```
gs.train(x=independent_vars, y=dependent_var,  
         training_frame=train_final,  
         fold_column=fold_column_name)
```

```
gbm Grid Build Progress: [#####]
```

Evaluate results &
model selection

```
gs.sort_by('mse', increasing=True)
```

Grid Search Results for H2OGradientBoostingEstimator:

Model Id	Hyperparameters: [learn_rate, ntrees, min_rows, max_depth]	mse
Grid_GBM_py_257_model_python_1446915311057_18_model_6	[0.01, 255, 5.0, 7]	1954.1
Grid_GBM_py_257_model_python_1446915311057_18_model_7	[0.01, 255, 10.0, 7]	1959.6
Grid_GBM_py_257_model_python_1446915311057_18_model_8	[0.01, 256, 15.0, 7]	1964.9
Grid_GBM_py_257_model_python_1446915311057_18_model_4	[0.01, 282, 10.0, 5]	2264.3
Grid_GBM_py_257_model_python_1446915311057_18_model_3	[0.01, 281, 5.0, 51]	2264.6

OPEN- SCIKIT PIPELINES

```
from h2o.transforms.decomposition import H2OPCA
from h2o.estimators.glm import H2OGeneralizedLinearEstimator
from h2o.model.regression import h2o_mean_squared_error
from sklearn.grid_search import RandomizedSearchCV
from sklearn.metrics.scorer import make_scorer
from sklearn.pipeline import Pipeline

pipeline = Pipeline([("pca", H2OPCA(k=2)),
                    ("glm", H2OGeneralizedLinearEstimator(family="gaussian"))])

params = {"pca_k": range(2, len(independent_vars)),
          "glm_alpha": [0, 0.5, 1],
          "glm_lambda": [1e-2, 3e-3, 1e-3, 3e-4, 1e-4]}

custom_cv = PreviouslyDefinedFold(train_final[fold_column_name])

random_search = RandomizedSearchCV(pipeline, params,
                                    n_iter=5,
                                    scoring=make_scorer(h2o_mean_squared_error),
                                    cv=custom_cv,
                                    random_state=42,
                                    n_jobs=1, refit=True)

random_search.fit(train_final[independent_vars], train_final[dependent_var])
```



Create Pipelines

Hyper-parameter Options

Cross validation strategy

Hyper-parameter
Search Strategy

Fit

KALMAN - POST PROCESSING

```
import pykalman as pyk

final_ensembled_preds = {}
pred_cols = [ name for name in predictions_df.columns if "predict" in name]
for unit in predictions_df.UnitNumber.unique():
    preds_for_unit = predictions_df[ predictions_df.UnitNumber == unit ]
    observations = preds_for_unit.as_matrix(pred_cols)
    initial_state_mean = np.array( [np.mean(observations[0]),-1] )
    kf = pyk.KalmanFilter(transition_matrices=a_transition_matrix,\n                          initial_state_mean=initial_state_mean,\n                          observation_covariance=r_observation_covariance,\n                          observation_matrices=h_observation_matrices,\n                          n_dim_state=n_dim_state, n_dim_obs=n_dim_obs)
    mean,_ = kf.filter(observations)
    final_ensembled_preds[unit] = mean
```

RESOURCES

- Download and go: <http://www.h2o.ai/download>
- Documentation: <http://docs.h2o.ai/>
- Booklets, Datasheet: <http://www.h2o.ai/resources/>
- Github: <http://github.com/h2oai/>
- Training: <http://learn.h2o.ai/>
- This presentation and associated Jupyter notebook
(look in 2016_01_21_MachineLearningAndKalmanFiltersForMachineHealth):
<https://github.com/h2oai/h2o-meetups/>

THANK YOU