# AWS-SAGEMAKER

## Overview:

Amazon SageMaker is a fully managed service that provides every developer and data scientist with the ability to build, train, and deploy machine learning (ML) models quickly.

SageMaker removes the heavy lifting from each step of the machine learning process to make it easier to develop high quality models.

* Label
* Build
* Train and Tune the data
* Deploy and Manage

For Build, train, tuning the data and deploying Sagemaker provides Amazon Sagemaker studio which is an Integrated Development environment for machine learning.

## Types of building algorithms:

The model building is usually done in three ways as per our requirement.

1. Amazon SageMaker provides some inbuilt algorithms like Linear Learner algorithm and XGBoost Algorithm. we can make use of these algorithms.
2. We can also build Custom Train-Inference pipelines such as sklearn models in sagemaker and can deploy it.
3. We can also build a Custom Docker Image Train-Inference Pipeline in Sagemaker.

Custom preprocess is common in all the above three before building a model and this can also be achieved by Sagemaker.

## 

## 

## 

## 

## 

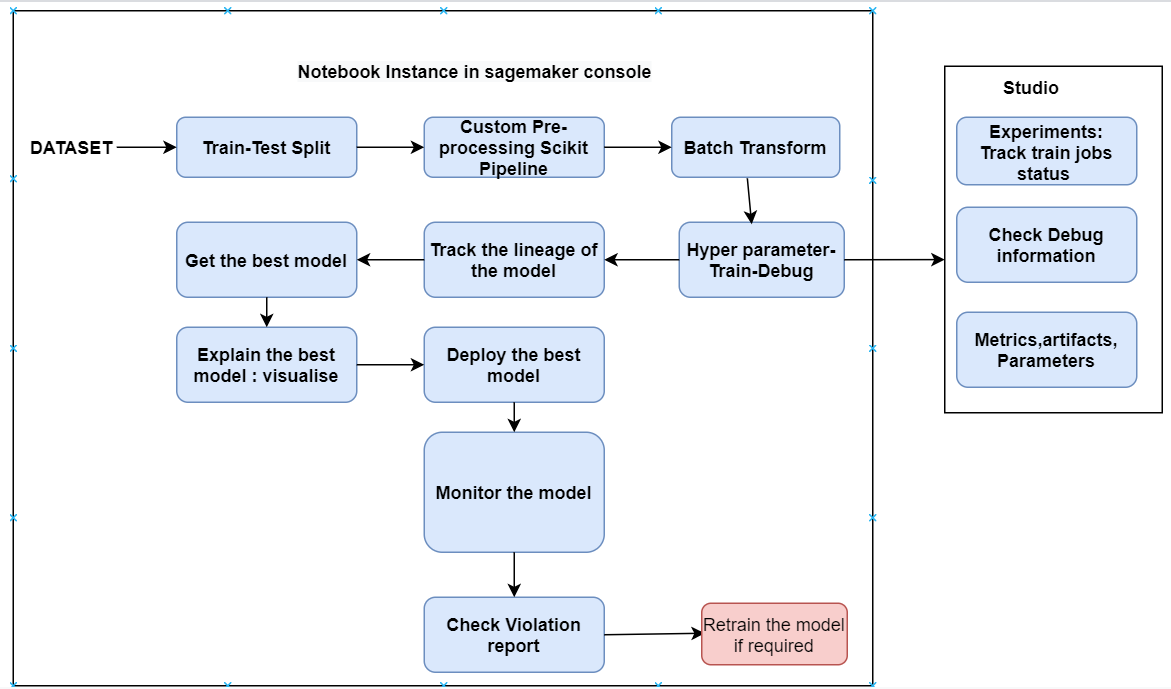
## 

## 

## 

## 1.Custom preprocess and Inbuilt-SageMaker model.

### 1.1. Workflow:

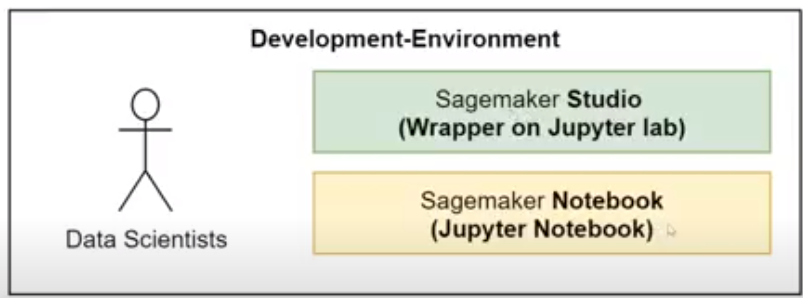


### 1.2. **Development environment :**

There are two environments in Sagemaker for Machine Learning.

1. Amazon SageMaker Studio, the first fully integrated development environment (IDE).

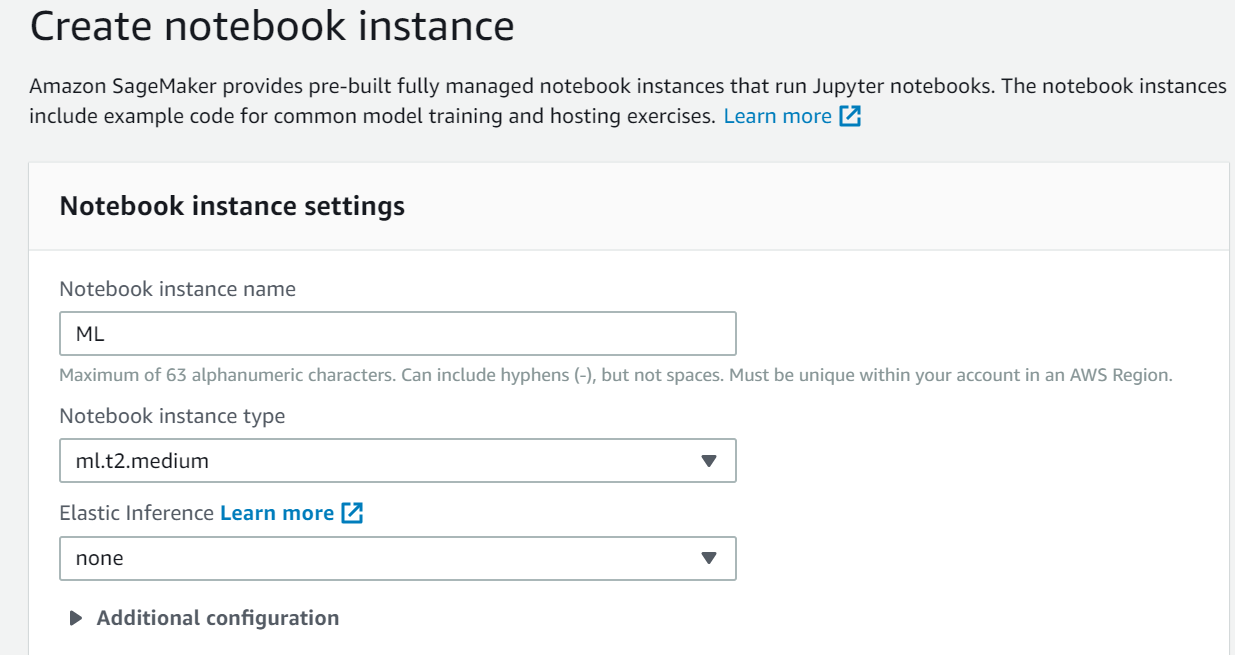
2.Notebook instance available in the sagemaker console.



we can create the notebook instance by selecting the required notebook instance type.

Preferable instance type to execute current code is: ml.m4.xlarge.

Kernel:conda\_tensorflow\_p36



There are some advantages of using notebook over studio:

* Suport all plugins
* Optimized cost
* Easy to maintain/retrying
* Inbuilt Docker available

### 1.3. **Dataset:**

Housing-Dataset.

* Each row represents one district. There are 10 attributes.
* longitude, latitude, housing\_median\_age, total\_rooms, total\_bedrooms, population, households, median\_income, median\_house\_value, and ocean\_proximity.
* There are 20,640 instances in the dataset.
* ‘total\_bedrooms’ attribute has only 20,433 nonnull values, meaning that 207 districts are missing this feature.
* All attributes are numerical, except the ocean\_proximity field.so further we are doing preprocessing.
* We are predicting median\_house\_value going further.

### 1.4. **Train-test-split:**

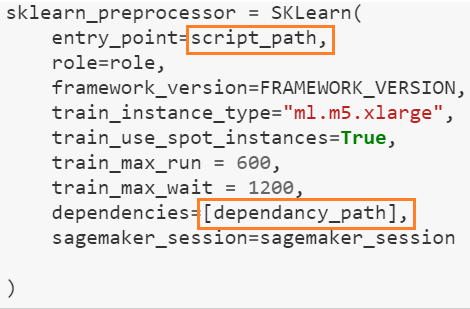
Random split

### 1.5. Preprocessing:

* We have used custom Scikit -learn Inference Pipeline for some pre-processing, customized it.
* So preprocessing steps are available in dependencies.py and sklearn\_pipeline.py(script\_path) files.

### 1.6. Batch Transformation:

There is a custom SKLearn processor where we can configure the entry point(script\_path) and dependencies\_path(we can add multiple dependencies).



**FIT and TRANSFORM(Train-data):**

When you are training a model, you will use the training dataset. On this dataset, we will use the Imputation and other preprocessing steps, calculate the values, and replace the blanks as per our requirement.

But when you fit this trained model on the test dataset, you don’t calculate the mean or median again. You’ll use the same value that you used on your training dataset.

For this, you’ll use the fit() method on your training dataset to only calculate the value and keep it internally in the Imputer.

Then, you’ll call the transform() method on the test dataset with the same Imputer object(as per custom). This way, the value calculated for the training set, which was saved internally in the object, will be used on the test dataset as well.

So fit the train and transform the train data.

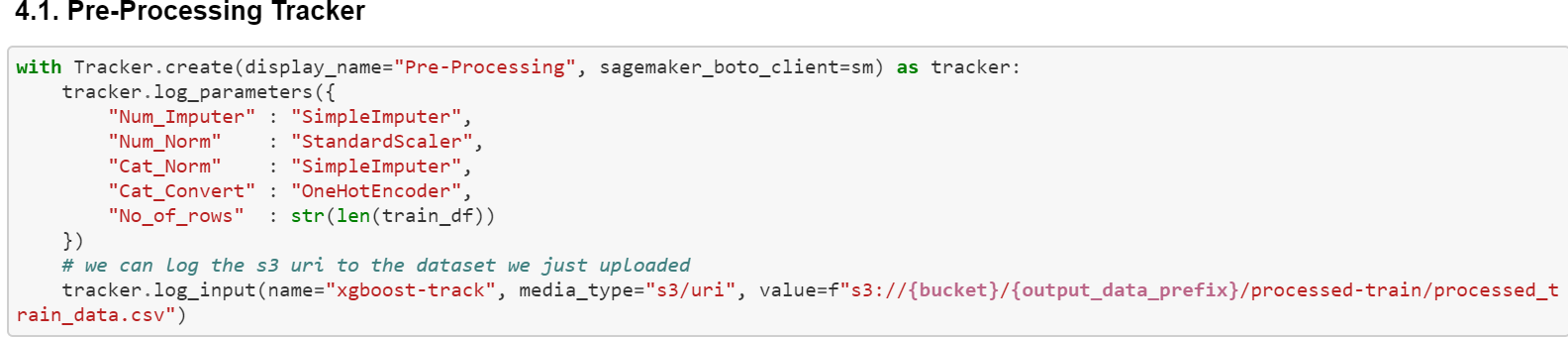
**Transform (Test-data)**:

you’ll call the transform() method on the test dataset.

We can store the processed train-data and test-data in S3

**Train-Track and Debug:**

We can track the train jobs by using Tracker.



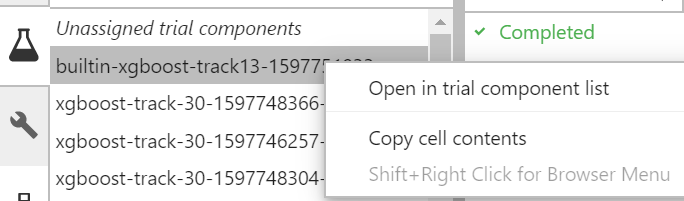
**Experiments**:

Amazon SageMaker Experiments is a new capability that lets you organize, track, and compare your machine learning training experiments on [Amazon SageMaker](https://aws.amazon.com/sagemaker/).

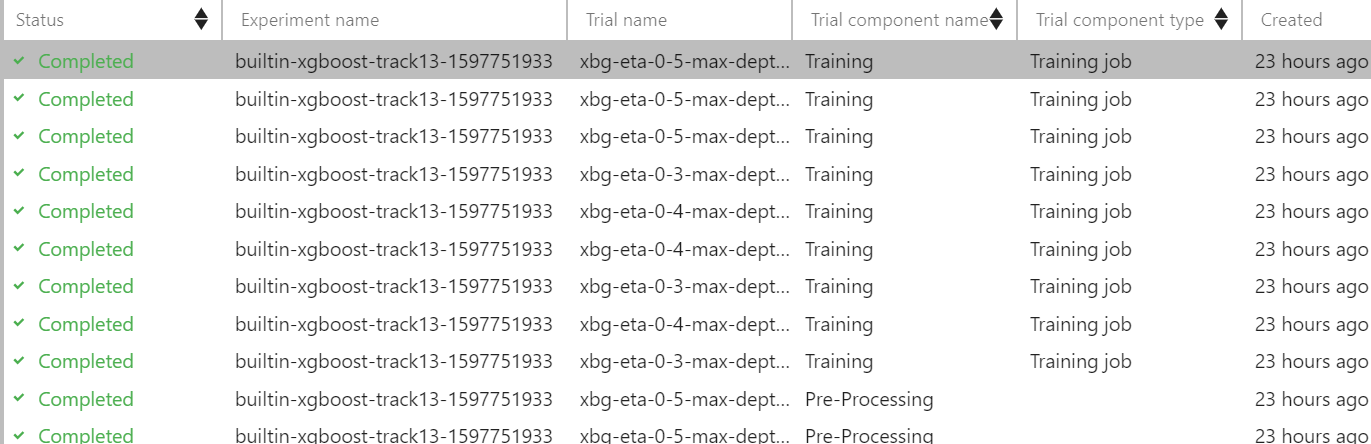
You can keep track of experiments in Amazon Sagemaker Studio.

1. We can find experiments  icon provided by the SageMaker at the left side of the studio.

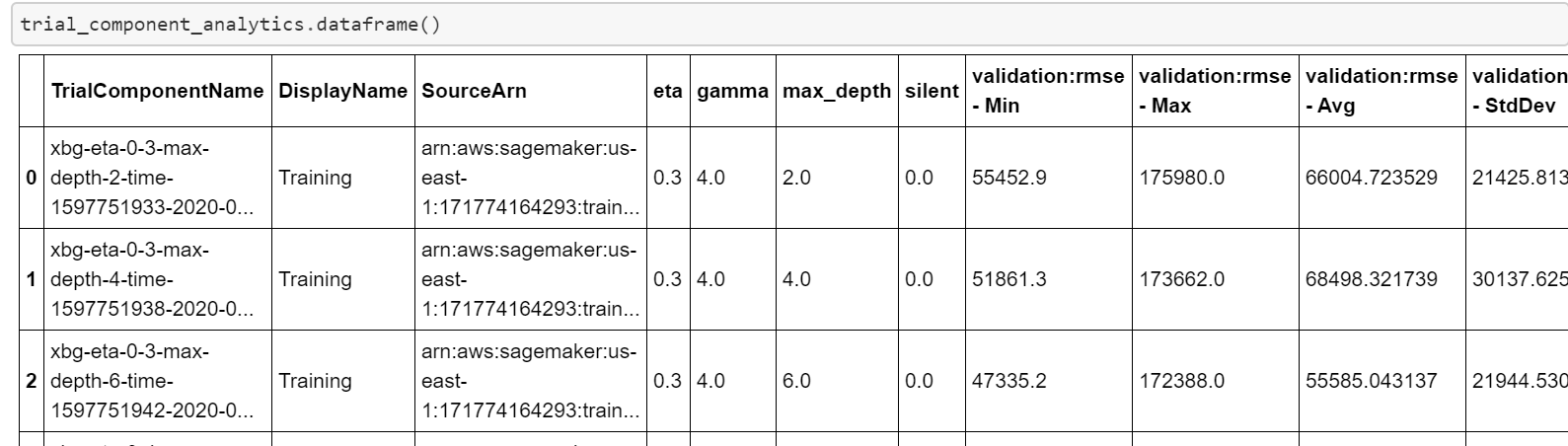
2. Right click on the particular experiment name we have created earlier and click on *Open in trial component list* for the status of multiple training jobs.



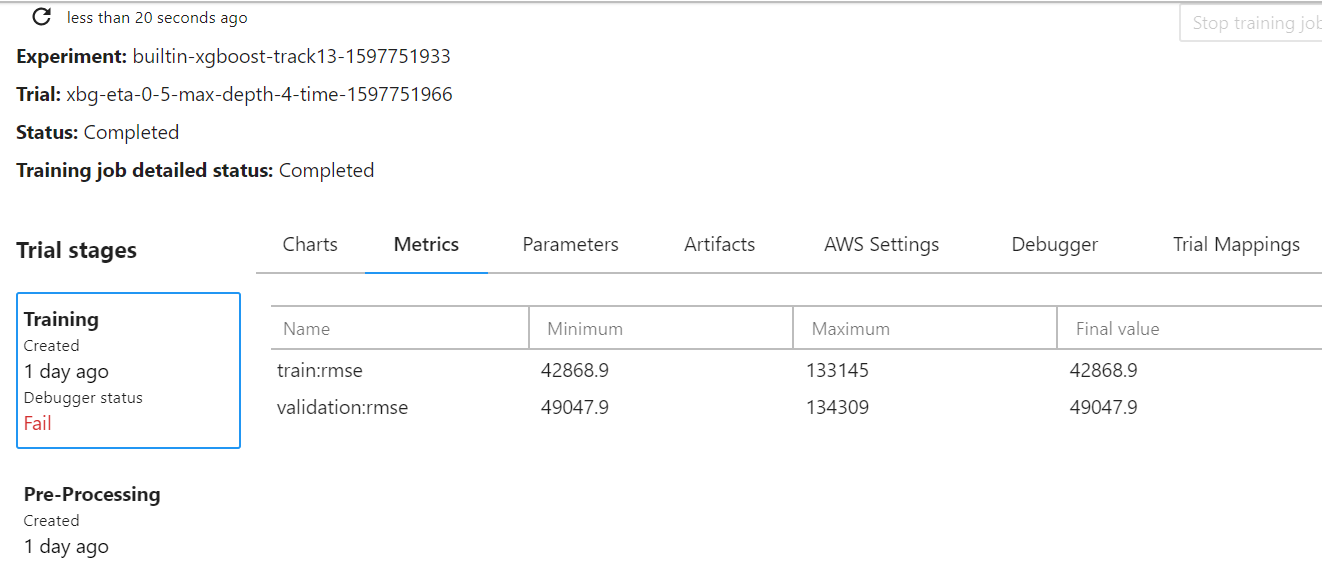
3. For all sets of hyperparameters it will create multiple train jobs. (status: in progress or completed)



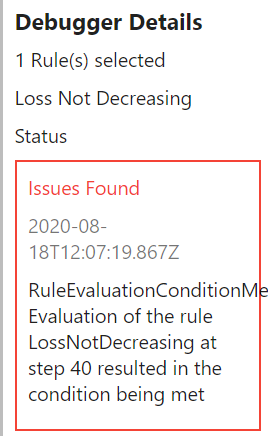
4. Once it has created it stores it as a dataframe internally and we can filter the best model as per the metrics available.



5. If we right click on particular trail job((ref 3.) and select trail details of it, we can able to find metrics, parameters (training as well as pre-processing), artifacts, debugger status as below:



6. [SageMaker Debugger](https://aws.amazon.com/sagemaker/) has the ability to capture tensors during training and store tensors of respective train jobs in s3 bucket and it will analyze whether the job is failed or not.



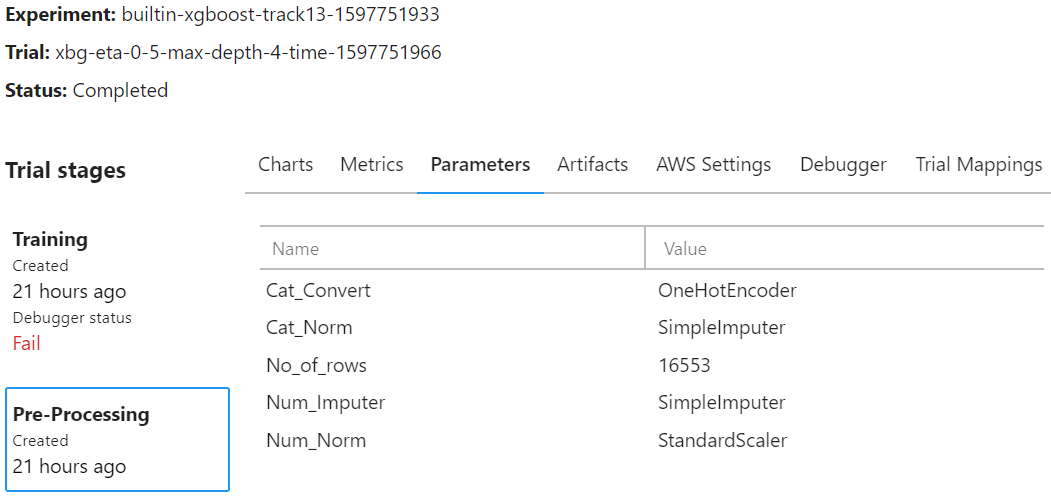
We generally use debugger status in case of neural networks and when jobs need to run for a long time. *LossNotDecreasing* will tell us loss was not decreasing during training process and we can stop the training job if required.



Using the [SageMaker](https://aws.amazon.com/sagemaker/) SDK and its estimators, you configure your training job as usual, passing additional parameters defining the rules we want [SageMaker Debugger](https://aws.amazon.com/sagemaker/) to apply.

Pre-defined rules are available for common problems such as exploding/vanishing tensors (parameters reaching NaN or zero values), exploding/vanishing gradients, loss not changing, and more. Of course, we can also write our own rules.

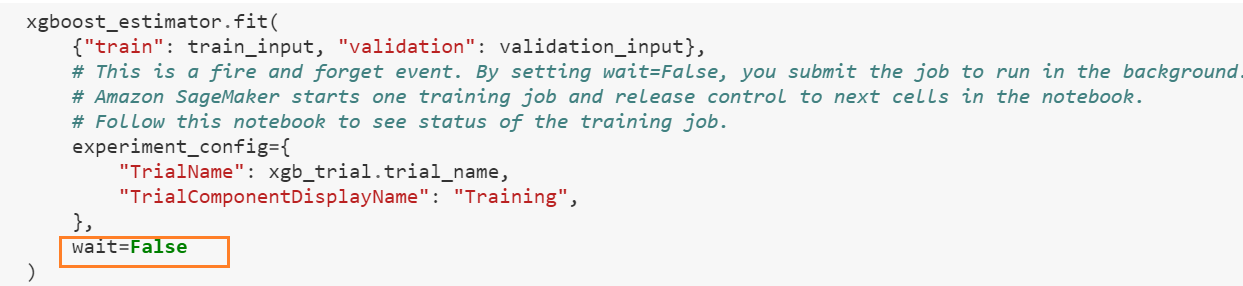
7. So we can also see Parameters used for pre-processing by tracker we have created earlier.



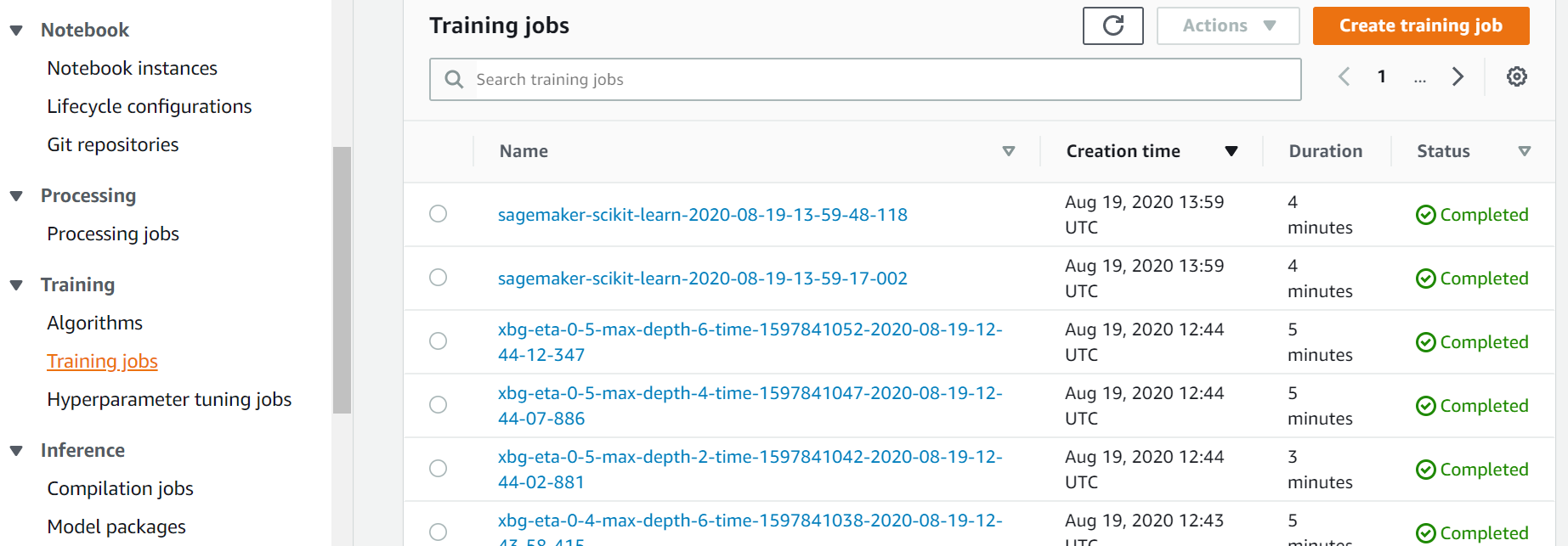
### 1.7. Sagemaker Console:

There are separate configurations available for tracking of training jobs, hyperparameter tuning jobs, processing jobs, endpoints etc which are available in SageMaker console.

Hyperparameter tuning jobs: It can be done through console or in studio. But both are independent of each other. The one which we are creating in the console can’t be tracked or logs cannot be seen in experiments. In order to see logs we need to create it manually.It will list down the training jobs for hyperparameter tuning run.



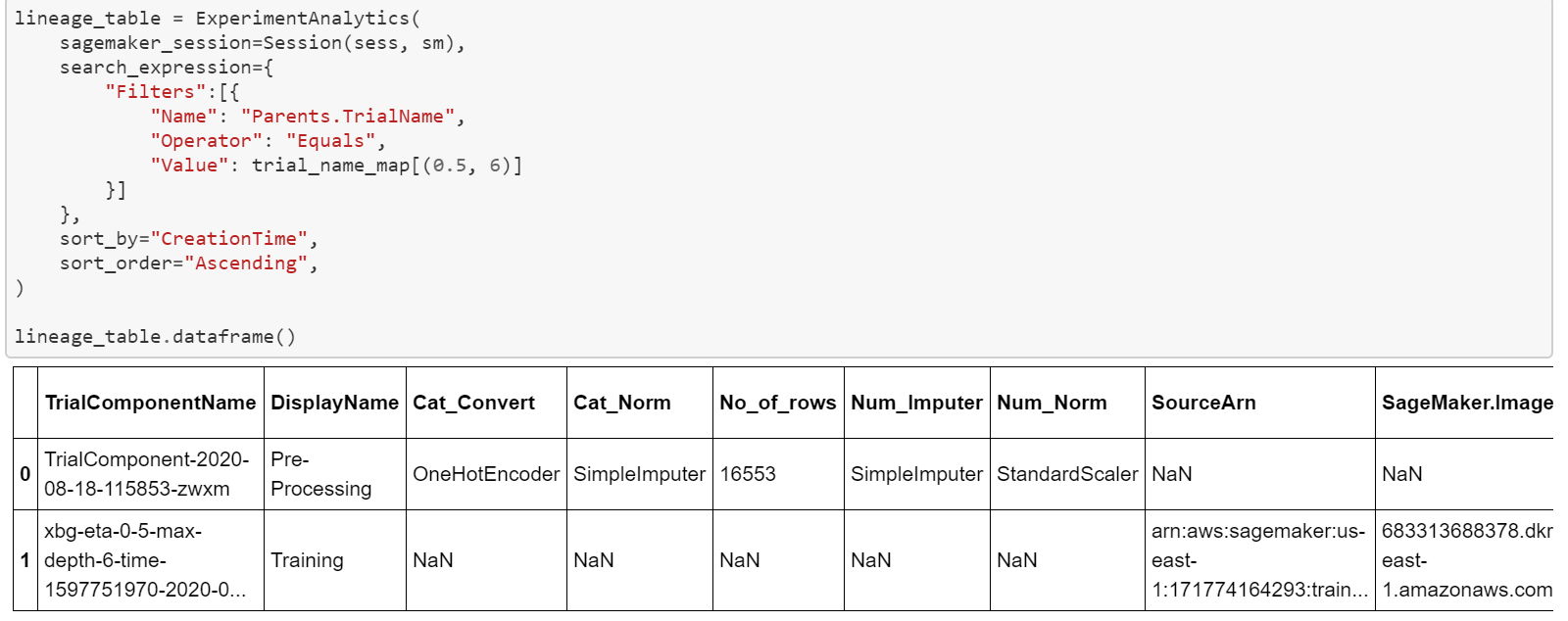
If we set wait=*False* during training, it won’t wait and will create jobs for each hyperparameter and can be seen in console as below.



### 

### 1.8. Sagemaker Lineage:

SageMaker Experiments enables tracking of all the steps and artifacts that went into creating and certifying a model, we can quickly track the lineage of a model when you are troubleshooting issues in production or auditing your models for compliance.



### 

### 1.9. Model Explainability with Amazon Sagemaker Debugger:

As we have used the Amazon SageMaker built-in *LossNotDecreasing*rule to monitor the metrics collection, we can use the tensor values which are generated for our best model to analyze.

Within a trial, a *step* represents a single batch of the training job. Each trial has multiple steps. A collected tensor has a particular value at each step. The tensor values are stored in the Amazon S3 location you specified.



We can list out our metric (eg train-rmse) how it’ s being scored for each step. we can do it for validation as well.

Since we have tensor values, we can use the explainability tool SHAP (SHapley Additive exPlanation) provided by the Amazon SageMaker Debugger for visualizations.

### 1.10. Visualizations:

We can visualize the feature\_importance, full\_shap, and average\_shap tensors captured during training.

Analysis includes arriving at global and local explanations of the model to understand how the individual features contribute to model predictions.

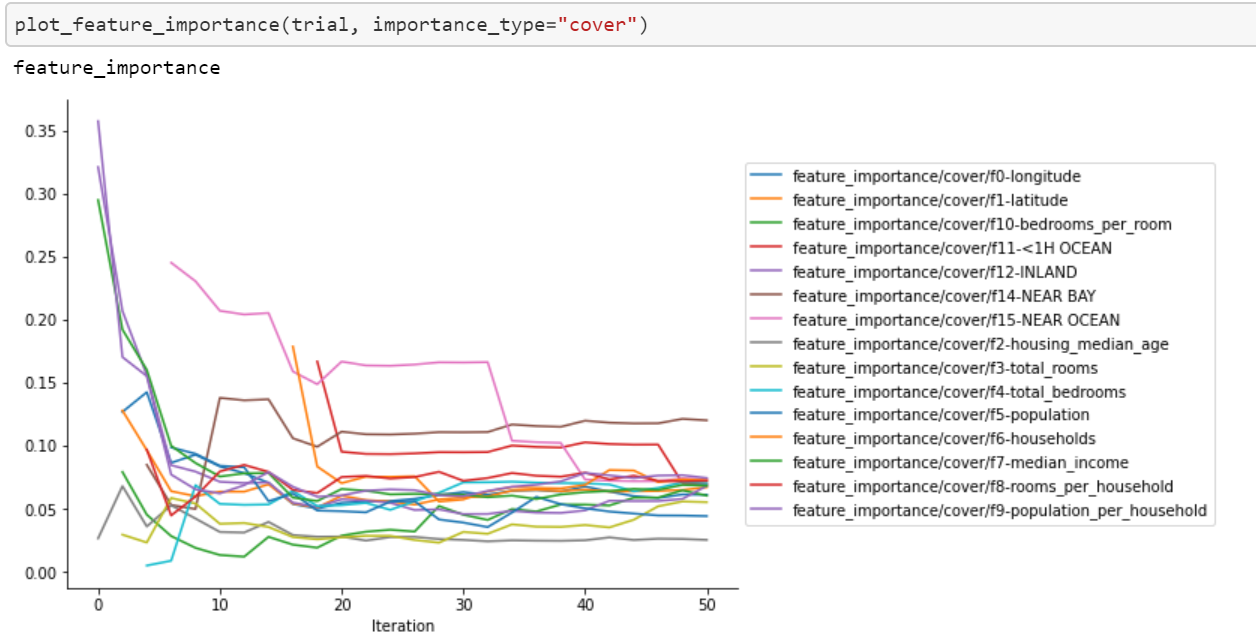
***feature\_importance***:

The actual names of the features aren’t included in the tensor names; they’re represented as f0, f1, and so on. This prevents sensitive feature names from showing up in analysis.

To view the average\_shap tensor value for f1, enter the following code:



You can also plot tensors collected for multiple features. For example, to plot the feature\_importance, enter the following code:



Similarly, you can plot the average\_shap tensor values collected for all features.

***Global Explanations*:**

Global explanatory methods allow you to understand the model and its feature contributions in aggregate over multiple data points. The following graph is an aggregate bar plot that plots the mean absolute SHAP value for each feature.

Specifically, the following plot(Figure A) indicates that the value of median\_income plays the most important role in predicting the median\_house\_price.

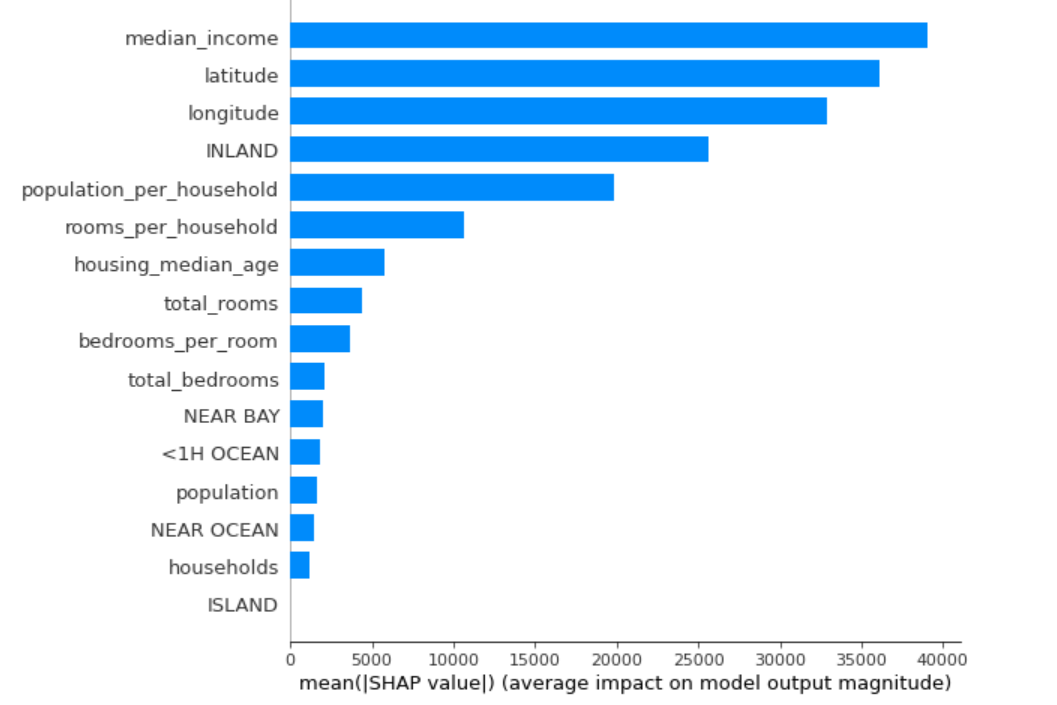


Figure A

*SHAP Summary Plot*:

The biggest difference of the plot (Figure B) with the regular variable importance plot (Figure A) is that it shows the positive and negative relationships of the predictors with the target variable.

The summary plot combines feature importance with feature effects.

Each point on the summary plot is a Shapley value for a feature and an instance.

The position on the y-axis is determined by the feature and on the x-axis by the Shapley value. The color represents the value of the feature from low to high. The features are ordered according to their importance.

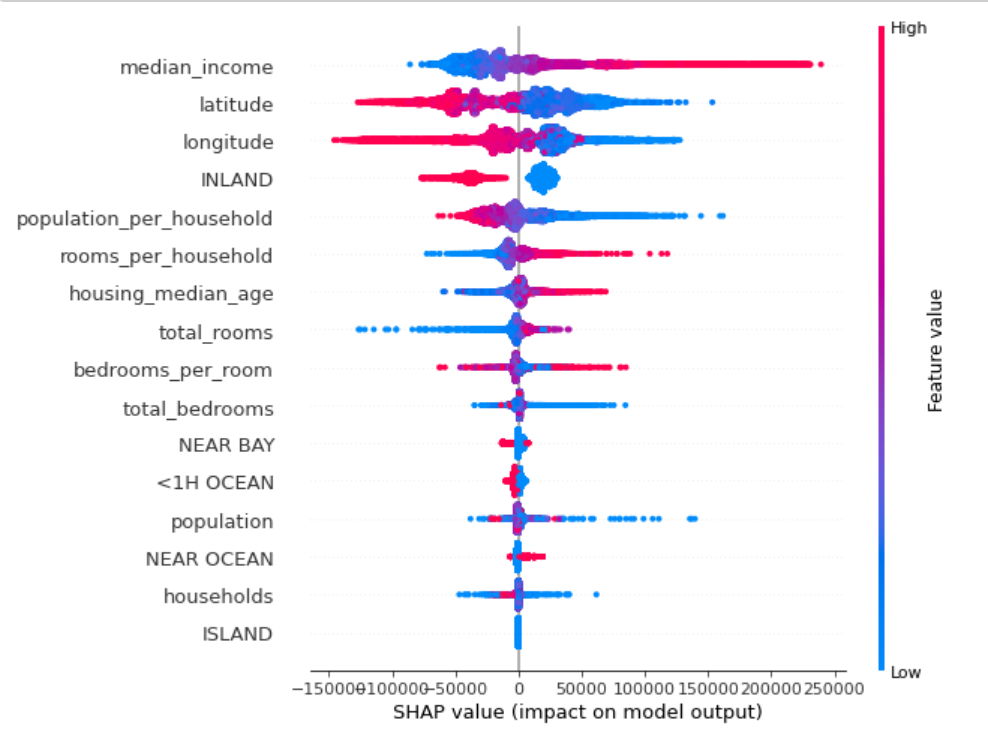


Figure B.

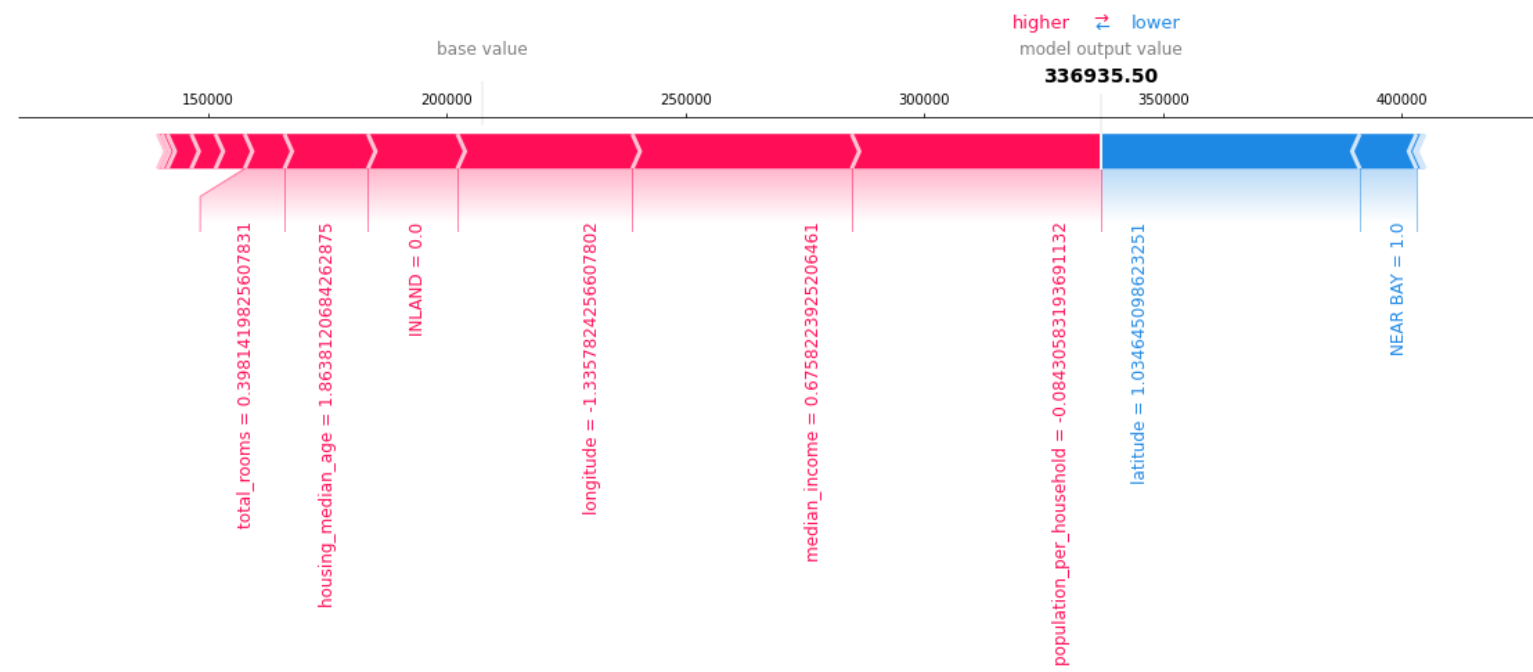
High value of median income, increases the predicted median\_house\_price.(which eventually leads to True predictions).

The color allows you to match how changes in the value of a feature affect the change in prediction.

***Local Explanations:***

Local explanations focus on explaining each individual prediction.

A force plot explanation shows how features contribute to pushing the model output from the base value (the average model output over the dataset) to the model output. Features pushing the prediction higher are in red; those pushing the prediction lower are in blue.

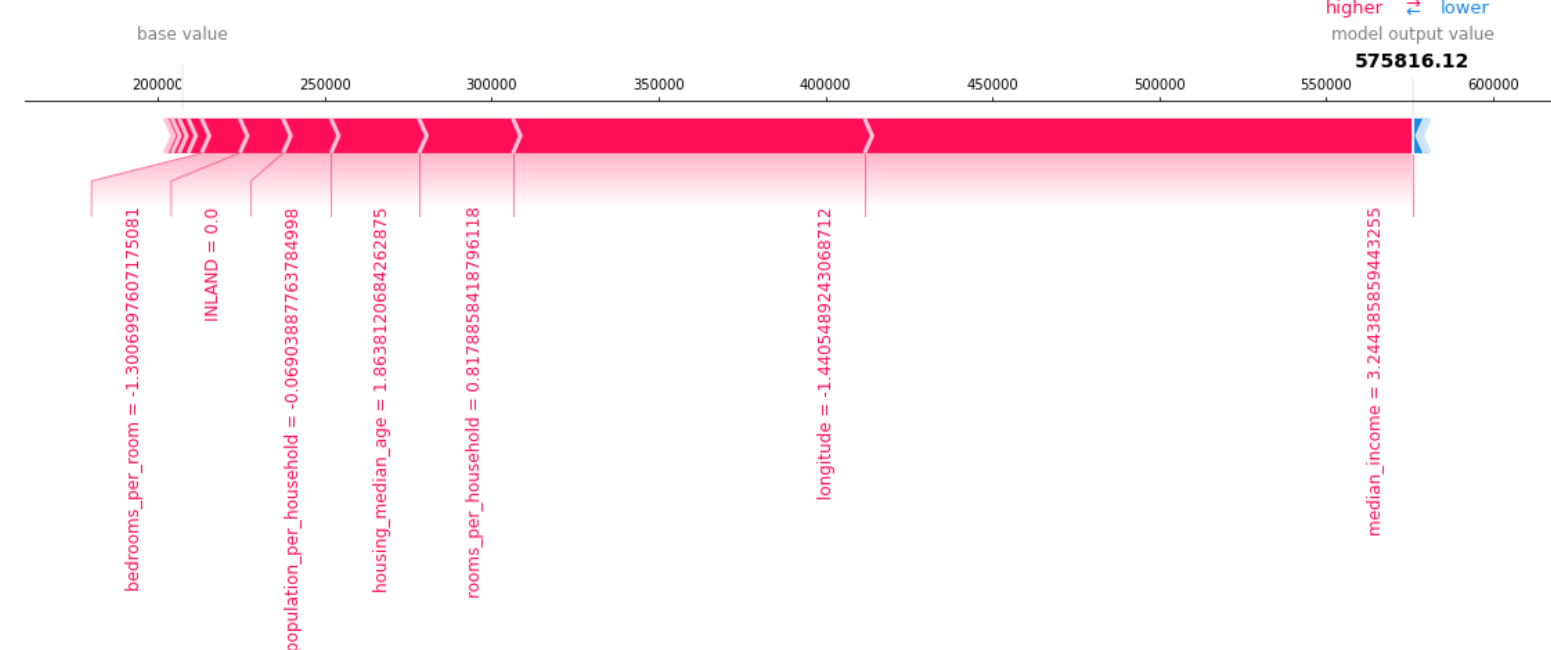


The above plot indicates that for a particular data point. The prediction probability is higher than the average , primarily because it has a higher value of median\_income.

***Outliers*:**

Outliers are extreme values that deviate from other observations on data. It’s useful to understand the influence of various features for outlier predictions to determine if it’s a novelty, an experimental error, or a shortcoming in the model.

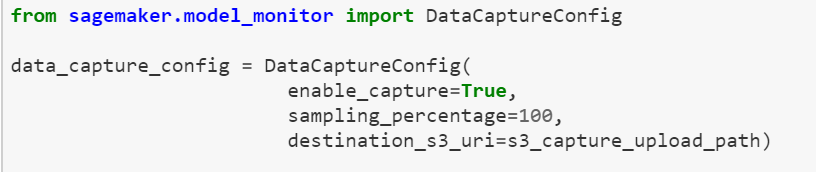
The following force plot shows prediction outliers that are on either side of the baseline value. The graph indicates that if “median\_income =3.244,housing median age=1.86” is probably an outlier to the group.



### 1.11. Deploy Endpoint:

After we train your model, we can deploy it to get predictions.

Enable data capture for monitoring the model data quality. We can specify the new capture option called *DataCaptureConfig*. You can capture the request payload, the response payload or both with this configuration. The capture config applies to all variants. We can now go ahead with the deployment.



### 1.12. Model Monitoring:

Sagemaker model monitor automatically watches over your production deployments.

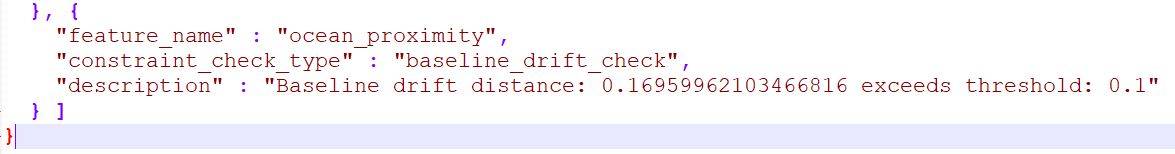
1. It collects (samples of data periodically from prediction endpoint).

2. Analyzes the data, looking for some key defining characteristics and comparing those of with training data.

3. If there is any deviation from training data baseline, model monitor issues an alert.

4. For example we are having model that predicts housing prices and the model interest rates(if feature avl) or ocean\_proximity changes..,

* Model becomes inaccurate(not right price predictions).
* Model monitor gives us an alert in cloudwatch which tells us ocean\_proximity feature which contributes towards housing price prediction is experiencing a drift and you model accuracy is going down.



* So we can go back and recalculate and retrain the model against the latest proximity feature data or the feature which is explaining drift as per your dataset.

Mode Monitoring works as shown below:

***Capture Data***: you can capture a configurable fraction of the data sent to the endpoint (you can also capture predictions if you’d like), and store it in one of your Amazon Simple Storage Service (S3) buckets. Captured data is enriched with metadata (content type, timestamp, etc.), and you can secure and access it just like any S3 object.

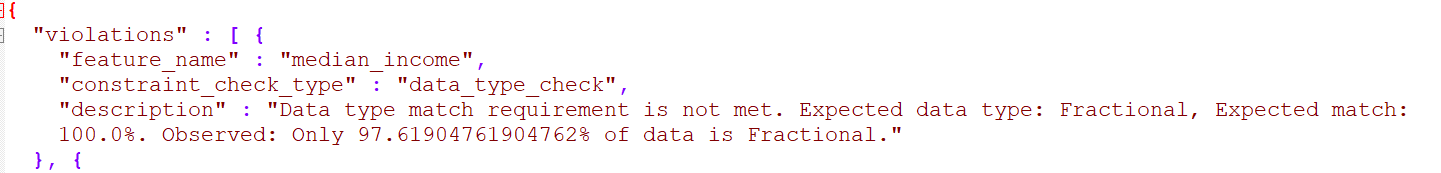
***Create a Baseline*:** You create a baseline from the data set that was used to train the model deployed on the endpoint (of course, you can reuse an existing baseline, too)

* Infer a schema for the input data, i.e. type and completeness information for each feature. You should review it, and update it if needed.
* Creates a baseline schema constraints and statistics for each feature.

***Schedule Monitoring Jobs***: Using these artifacts, the next step is to launch a monitoring schedule, to let SageMaker Model Monitor inspect collected data and prediction quality.

* Reports are periodically pushed to S3
* The reports contain statistics and schema information on the data received during the latest time frame, as well as any violation that was detected.

Generally the violation report looks like below: If there is any datatype mismatch ,it is generated in the violation report and tells how much quantity of data is meeting expected match.



For a better model monitoring steps you can go through the model monitoring notebook provided in github [link.](https://github.com/tigerrepository/MLE-Playground/tree/feat/aws-sagemaker)

### 1.13. Pros and Cons using this approach:

**Pros:**

* Model explainability, SHAP.
* Amazon SageMaker Debugger- automatically identifies complex issues developing in machine learning (ML) training jobs.

**Cons:**

* we cannot define our own metrics (custom metrics is not possible)