**Attrition Model Documentation**

Table of Contents

[Summary 2](#_Toc171283894)

[Architecture 3](#_Toc171283895)

[Gitlab 3](#_Toc171283896)

[Folder Structure 4](#_Toc171283897)

[.gitlab-ci.yml 5](#_Toc171283898)

[Pipeline 5](#_Toc171283899)

[Config 5](#_Toc171283900)

[Training - 5](#_Toc171283901)

[Data Pipeline 6](#_Toc171283902)

[Final Features 6](#_Toc171283903)

[Training 6](#_Toc171283904)

[Requirements 7](#_Toc171283905)

[Lifecycle Configuration 7](#_Toc171283906)

[train.py 8](#_Toc171283907)

[Hyperparameter Tuning 8](#_Toc171283908)

[Model Outputs 8](#_Toc171283909)

[Deployment/Model Serving 8](#_Toc171283910)

[Inferencing 9](#_Toc171283911)

[Scoring 9](#_Toc171283912)

[Cloudwatch 9](#_Toc171283913)

[Notifications 9](#_Toc171283914)

[Secrets 9](#_Toc171283915)

# Summary

Medtronic commissioned a POC (proof of concept) to investigate the underlying causes of customer attrition for its diabetes pump device and to build a predictive model around the consumption of device consumables within a 6-month dtime span since a customer’s last purchase. To meet this goal, multiple sources of business data were extracted and transformed into features for a Bayesian model which proved successful in identifying causal factors for attrition as well as having a high performance on identifying likely attritioners. With the conclusion of the POC, Medtronic wanted to expand upon this work by productionizing the attrition model on AWS and creating an MLOPs framework to productionize future models created by its internal data science team. This entailed a thirteen week effort to gather new requirements, build out the cloud infrastructure, and test an end to end pipeline to operationalize this vision. Among the new requirements included utilizing Sagemaker for training and serving models, enabling the reading and writing of data to and from Snowflake, adapting the model to the XGBoost framework, and incorporating CICD tools such as Gitlab and Artifactory to automate deployment as much as possible.

# Architecture

A diagram of a company

Description automatically generated

Gitlab is the starting point for the CICD pipeline as it serves as a code repository and enables us to package and build the codebase for use in the cloud. Cloudformation allows us to deploy our tech stack to a new AWS account. For the training/serving pipeline, we have utilized an EC2 instance as what’s known as a Gitlab Runner, to package the training code into a gzip file that syncs with a S3 bucket. There is currently a ticket with AWS support to allow for direct Gitlab connectivity to a Sagemaker notebook, which would obviate the need for this architectural piece.

We utilize a Sagemaker notebook and the XGBoost API to call on this gzip file in S3 to execute a training script and also deploy the trained model into a serving environment. For inferencing, we are utilizing an inferencing container image that is built by Kaniko and ultimately sits in Artifactory. Batch enables us to call on this image and Cloudwatch triggers allows us to initiate a cron expression to schedule the image script. Both the train gzip file and inferencing image tap into multiple services such as Snowflake, Secrets Manager, and SNS.

# Gitlab

Gitlab is the central platform for our CICD pipeline as it allows us to host our codebase and update with changes for continuous integration, and package and build our codebase to different environments for continuous deployment.

## Folder Structure

The code repository on GitLab is under the Diabetes\_ML team and is called attrition\_model. The file structure found within that repository is shown below:

├── common/

│   ├── scripts/

│   │   ├── complaint.py

│   │   ├── demo.py

│   │   ├── fulldata.py

│   │   ├── merge.py

│   │   ├── sales.py

│   │   ├── score.py

│   │   ├── startright.py

│   │   ├── touchpoint.py

│   │   ├── warranty.py

│   ├── sql\_scripts/

│   │   ├── create\_complaint\_feature\_int\_table.sql

│   │   ├── create\_demo\_feature\_int\_table.sql

│   │   ├── create\_hist\_table\_generic.sql

│   │   ├── create\_modeling\_dataset.sql

│   │   ├── create\_sales\_feature\_int\_table.sql

│   │   ├── create\_scores\_table.sql

│   │   ├── fetch\_startright\_feature\_data.sql

│   │   ├── fetch\_touchpoint\_feature\_data.sql

│   │   ├── fetch\_warranty\_feature\_data.sql

│   │   ├── insert\_table\_generic.sql

│   │   ├── truncate\_table\_generic.sql

│   ├── utils/

│   │   ├── config\_checker.py

│   │   ├── database.py

│   │   ├── notifications.py

│   │   ├── time\_bucket.py

│   ├── \_\_init\_\_.py

│   ├── config.yaml

├── inferencing/

│   ├── inference.py

│   ├── setup.cfg

│   ├── setup.py

├── training/

│   ├── requirements.txt

│   ├── train.py

├── .dockerignore

├── .gitignore

├── .gitlab-ci.yml

├── ct.yaml

├── dev\_ct\_params.json

├── Dockerfile.inferencing

├── prod\_ct\_params.json

├── README.md

├── replace\_model.ipynb

├── train\_nb.ipynb

The **common** directory contains files that are utilized by both the training and inferencing pipelines. The **scripts** subdirectory contains Pandas logic to create features and the **sql scripts** subdirectory contain table schemas for all the intermediate and final tables, as well as fetch queries to fill in these tables with the requisite data. The **inferencing** and **training** directory contain files to download the necessary Python libraries as well as a main script titled inference or train.py. Cloudformation templates are directed by the **ct.yaml**, **dev\_ct\_params.json**, and **prod\_ct\_params.json** files.

## .gitlab-ci.yml

This file is executed whenever one pushes code to the remote repository, automating the packaging and building of code to various deployment environments. It is split up into 4 stages, broken up between training (for dev and prod) and inferencing.

The training portion packages the code into a file called **src.tar.gz**, which is synced with a s3 bucket, while the inferencing portion calls upon the **Dockerfile.inferencing** and builds a container image directed toward an Artifactory repository.

## Pipeline

Finally, one can check if the CICD pipeline was successful by navigating to the CI/CD button on the Gitlab console. As seen below, you can see if the entire pipeline passed or failed along with the current commit that launched the CICD pipeline. You can also restart or cancel a current pipeline as needed.

A screenshot of a phone

Description automatically generated

# Config

The **config.yaml** file is a reusable component which enables one to adapt changing parameters to new projects and environments, so one doesn’t have to hardcode these elements. It is located in the **common** directory. Any changes to this file will flow down to both the training and inferencing scripts. The file is essentially a large, nested dictionary mainly broken into training and inferencing.

# Training -

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Parameter Name** | **Type** | **Possible Values** | **Min** | **Max** | **Comments** |
| sampling | String | down, up |  |  | To use under or oversampling during training |
| split | Dict |  |  |  | How to create the training set |
| mode | String | last\_n\_months, randomized |  |  | Within split, method for creating training set |
| months | Integer |  | 1 |  | Within split, utilized if mode == last\_n\_months. Determines number of months in test set. |
| start\_month | String | Regex: ^[0-9]{4}-[0-9]{2}$ |  |  | Must be captured by Regex. |
| max\_month | String | Regex: ^[0-9]{4}-[0-9]{2}$ |  |  | Must be captured by Regex. Upper bound on training set |
| features | List |  |  |  | Features to keep in training set |
| rename\_cols | Dict |  |  |  | Key in dict is the current column name and the value is the renamed column name |
| drop | List |  |  |  | Columns to drop |
| dummy\_cols | List |  |  |  | Columns to apply one hot encoding to |
| target | String |  |  |  | Name of target variable |
| hyperparameters | Dict |  |  |  | Which hyperparameters to tune |
| optimize | String | auc, f1, accuracy |  |  | Metric to optimize |
| num\_tuning\_rounds | Integer |  | 1 |  | Number of trials |
| eta | Dict |  |  |  | Step size shrinkage |
| min | Integer |  |  | 0 | Within eta, min value |
| max\_depth | Dict |  |  |  | Max depth of a tree |
| min | Integer |  | 1 |  | Within max\_depth, min value |
| max | Integer |  | 1 |  | Within max\_depth, max value |
| subsample | Dict |  |  |  | Subsample ratio of training set |
| gamma | Dict |  |  |  | Min loss required for further split |
| min\_child\_weight | Dict |  |  |  | Min sum of weight in child node |
| reg\_alpha | Dict |  |  |  | L1 regularization on weights |

# Data Pipeline

Snowflake is the data warehouse. The Snowflake team has migrated the necessary business and device data to Snowflake to enable the current project. When deployed to production, forwarded ddls to the Snowflake team to create the tables for the model to consume as they have to tie read write privileges.

Data Model and Data Versioning

A diagram of a company

Description automatically generated

## Final Features

In total, there are 23 features utilized by the deployed model. The target variable is **IS\_ATTRITION\_C**, which is whether a customer hasn’t purchased a product consumable in the past 6 months.

# Training

All of the files specifically for training are in the **training** directory except for **train\_nb.ipynb** and **replace\_model.ipynb**.

## Requirements

The **requirements.txt** file indicates which Python libraries to download.

## Lifecycle Configuration

The lifecycle configuration script in **attrition-notebook-config** has to be manually installed as it provides additional environment variables and indicates which Python kernel for the Notebook to use. Please note that you if you change the file, you will have to stop and restart the corresponding Sagemaker notebook. Currently, it has a couple environment variables utilized by the notebook:

* secret\_name
* region
* account
* code\_S3 (s3 bucket where training code is stored)
* endpoint\_name
* subnets
* sgid

creates a new endpoint config tied to a previously trained model and replaces the model behind the current endpoint.

## train.py

This file is the main entrypoint for the training pipeline. In the file, it accomplishes the following actions:

* creates and populates **PA\_MODELING\_DATASET** by creating intermediate feature tables as well as merging them
* splits the train and test set
* drop and rename columns as needed
* perform hyperparameter tuning
* utilizes XGBoost for classification
* saves model outputs to s3
* send job notifications if job succeeds or fail

## Hyperparameter Tuning

While Sagemaker can do hyperparameter tuning, it is not able to return all of the required metrics without the use of a container image. We used the following hyperparameters

* eta
* gamma
* max\_depth
* min\_child\_weight
* num\_round
* objective
* reg\_alpha
* subsample

## Model Outputs

The model outputs are saved in the model directory as supplied by Sagemaker built in variable **SM\_MODEL\_DIR**. One can go to S3 after training is complete and unzip the file to view the contents. This includes:

* attrition.model – saved xgboost model
* confusion\_matrix.html – confusion matrix for test set trained by best model

## Deployment/Model Serving

Sagemaker allows us to not worry about setting up our own model serving infrastructure (Flask, nginx, gunicorn) for real time inferencing. All we have to do is define four functions below in the train.py file:

* model\_fn – load attrition.model from model directory
* input\_fn – deserialize data from json and transform into numpy array
* predict\_fn – transform input into xgb.Matrix datatype and predict
* output\_fn – serialize predictions into json form as output of scoring engine

# Inferencing

All of the files specifically for inferencing are in the **inferencing** directory.

## Scoring

The file **score.py** which is found in the common/scripts is called in **inference.py**. The file taps into attrition-model endpoint for model and returns probabilities and binary output value (0/1), creates forecasts, writes inference profiling to s3 bucket, and writes data to **PA\_INFERENCE\_RESULTS** table

# Cloudwatch

AWS CloudWatch provides logging capabilities for any stdout your code produces. The log groups of interest for the training and inferencing pipelines, along with the type of information captured by each, is shown below:

|  |  |
| --- | --- |
| **Log Group** | **Information in Logs** |
| /aws/sagemaker/Endpoints/attrition-model | Logs for model serving / deployment environment |
| /aws/sagemaker/TrainingJobs | Logs for model training |
| /aws/batch/job | Logs for AWS Batch job executing the inferencing pipeline |

Cloudwatch triggers allow us to schedule the Batch job using a cron expression, which is currently set up at 9:00 EST/13:00 GMT.

# Notifications

AWS SNS (Simple Notification Service) enables us to send email messages to subscribers to the **job\_notification** topic, indicating the success or failure of the training and inference pipeline. The feature can be found in the **notifications.py** file under the **common/utils** directory.

# Secrets

AWS Secrets allows us to define environment variables for sensitive information and pass them securely to our codebase.

* + user
  + account
  + warehouse
  + database
  + schema
  + role
  + snowflake\_passphrase
  + snowflake\_pk\_secrets\_name (match the name of the 2nd secret below)