# Multimodal Data Algorithm

## Implementation and Usage Guide

*Viswanath Chadalapaka*

# Contents

Overview

- Summary
- Use Cases
- Implementation Details
- Deployment and Usage Instructions

Shortcomings and Future Improvements

- For this Project
- For SageMaker

Scripts and Links

# Overview

## Summary

The Multimodal Data Algorithm is an AWS SageMaker Algorithm built to facilitate the efficient and timely training on and inferencing for the classification of multimodal data. Currently, the algorithm does not support regression, and only supports training data passed in as a CSV. The algorithm uses the train data CSV's headers to determine the modality of each column of data. As it stands, the algorithm supports four main categories of data that must each be formatted and entered into the training CSV file as follows:

- **Text** features must be placed under headers ending with "_text". While text data may include newline or new paragraph characters in the training data, data passed in for inferencing *cannot* contain these characters due to the limitations of SageMaker, so it may be a good idea to strip these from your text columns before using the algorithm. It is alright for text to contain any other characters, including non-ASCII characters. A simple script to perform such a task is provided near the end of the document. Text data that is missing will be imputed as an empty string.

- **Numerical** features must be placed under headers ending with "_num". Numerical values that are missing will be imputed as 0 (*not* the mean).

- **Categorical** features must be placed under headers ending with "_cat". While the algorithm supports any categorical variables, it is a good idea to remove categorical features that are mostly unique for a row, as these will have a significant negative impact on the algorithm's performance. Also, it should be noted that using a category with the value "unknown" has extra meaning to the algorithm, as missing categorical features will be imputed as "unknown".

- **Image** features must be placed under a header ending with "_image". Images must be byte-encoded in base 64. Currently, only JPEG images are supported. The encoded string will be processed by a script provided near the end of the document, so use of the script to ensure the proper processing of images may be helpful. Empty image features will be imputed as a 224x224 black square.

Some additional rules related to model usage are that the label provided with the train data must have the header "label", and the IDs provided with both the train and test datasets must be under the header "ID". The test dataset should not have any headers, and the ordering of the feature columns in both the train and test datasets must be the same.

Currently, the inferencing portion of the algorithm **only** supports batch transform jobs, with data split type as "Line".

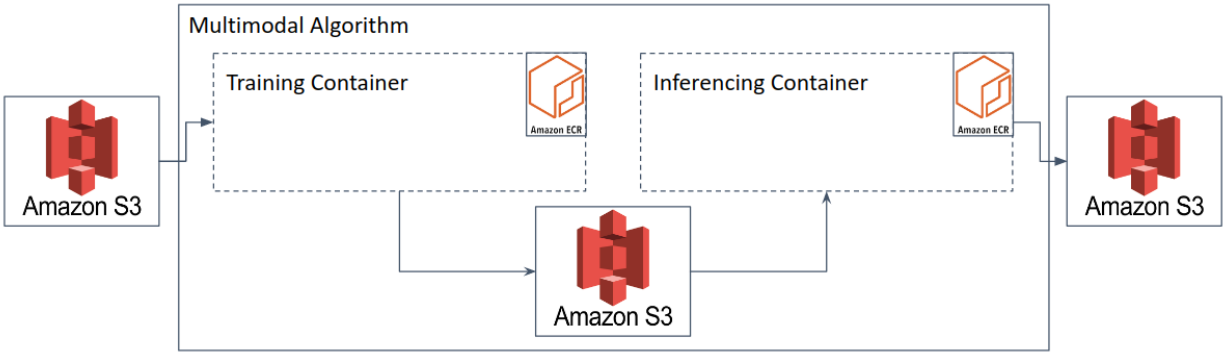
## Use Cases

The main use cases of multimodal classification, in the context of consumer goods sellers, are to provide better product suggestions and a better customer experience. Scholastic,

for example, has a need to classify books into genres based on the cover, description, and page length. Netflix categorizes movies and TV shows into genres based on similar information. A secondary use case is also the labelling and taking down of "restricted products" – in the case of Amazon, for example, this would include the accurate labelling of items like firearms, to aid in their removal from the market.

**Implementation Details**

Since the Multimodal Algorithm solution is a SageMaker algorithm, when loaded by a training job in SageMaker, it will take training input from an S3 bucket. After the training job completes, the algorithm also uploads model artifacts to an S3 location. Finally, when inferencing, the algorithm refers back to the model artifacts stored to produce and store predictions in a 3rd S3 location. In the event that the training job successfully completes, but the inferencing job cannot successfully be started, a link to a Jupyter notebook is provided near the end of the document for the purpose of inferencing outside of the containerized algorithm solution using the model artifact output (the conditions under which inferencing job failures can occur are laid out in the "shortcomings" section).

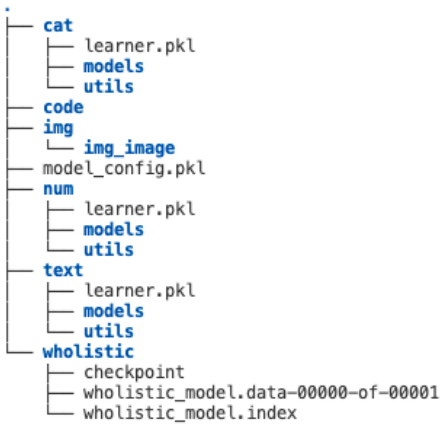



While the following information about the training stage of the model is not nearly enough to make use of the model artifact output, it is nevertheless important in determining the strengths and weaknesses of the model (in the "shortcomings" section), and will give some context to the Jupyter notebook linked at the end of this document. The training stage roughly performs the following steps, in order:

1. First map *n* user-defined categories to integers 0 -> n-1 (e.g. a, b, c -> 0, 1, 2) and map each label given to the integers. This set of labels will be called the mapped labels.
2. Split the data into text, numeric, categorical, and image features.
3. Generate "modality-specific" labels. "Modality-specific" labels are generated for each modality except for the image modality, for which a set of labels is generated for each image feature. These are labels that come from one of two sources:
    a. The mapped labels (1 -> n-1) from step 1, in the event that the modality provided enough information on that row (not too many NaNs)
    b. OR, a new integer label n, in the event that not enough information was provided by the modality for the given row, caused by a large number of empty features for the given modality in the row
4. Transform all images to numerical vectors by running them through a pretrained resnet50 image classification model. As stated earlier, empty images are treated as 224x224 black squares.

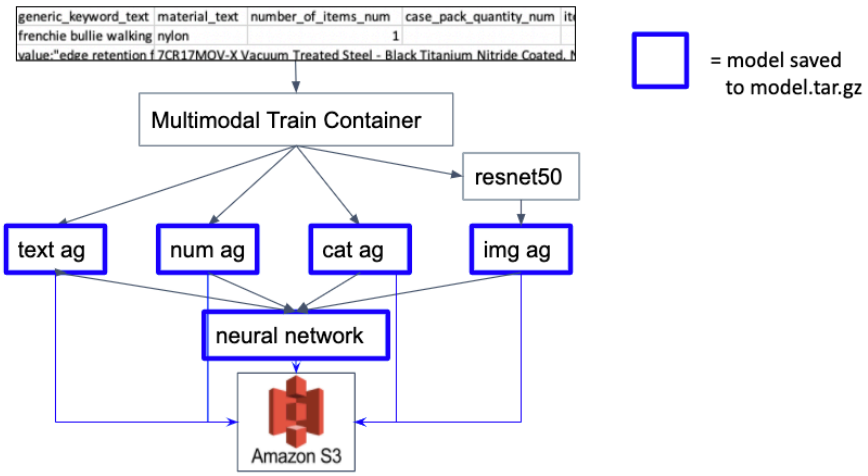
5. For each modality, start an Autogluon job with the labels set to the “modality-specific” labels generated earlier.
6. Run all train data through the models generated by the Autogluon training jobs to obtain probability distribution size n vectors from training data to “modality-specific” labels.
7. Train a neural network to map a concatenation of modality-specific model outputs vectors to mapped labels 1 -> n-1 (without n). This network joins together all of the individual modality’s predictions to make a single prediction.
8. Save each modality’s Autogluon model to the models folder at [MODEL_DIRECTORY]/modality/, and save the neural network to [MODEL_DIRECTORY]/wholistic/.
9. Save additional model information, such as column names, to [MODEL_DIRECTORY]/model_config.pkl.

After completion of a training job, the model artifacts are compressed into a model.tar.gz file and stored in a specified S3 location. Depending on the data received, the uncompressed model file structure looks something like the following tree output:

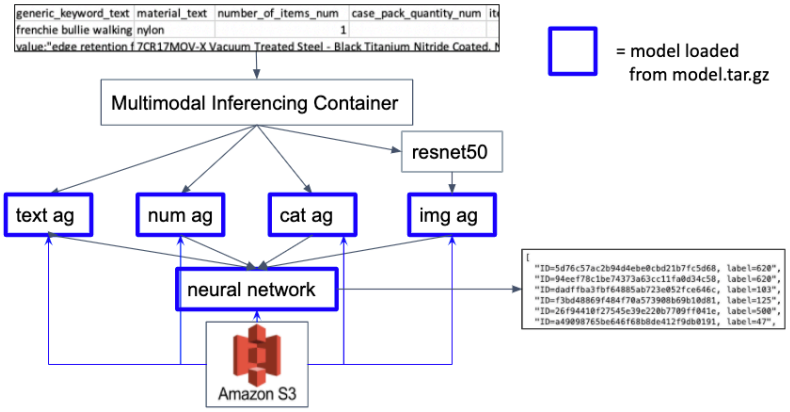



Each of the folders “cat”, “img”, “num”, and “text” are Autogluon training job outputs, as mentioned in the training steps earlier. Not all of these folders will exist, depending on the training data passed in. “wholistic/wholistic_model” is a load point for the weights of a tensorflow-based neural network with an input layer, 2 hidden layers, and an output layer. The sizes of these layers are based on the way in which the “modality-specific” labels were generated for the Autogluon jobs, and the number of user-defined categories. All of this information is also saved in model_config.pkl.

The following flowchart demonstrates in a nutshell how the algorithm trains using a given dataset:




The following flowchart demonstrates in a nutshell how the inferencing algorithm uses the S3 model artifact to make inferences:




**Deployment and Usage Instructions**

As the algorithm is not yet fit for production (the reasons for this will be made clear in "shortcomings"), it has not been pushed to the AWS Marketplace (although it has passed the validation criteria necessary for publishing). Therefore, this algorithm must be built from the following repository: https://github.com/Derposoft/multimodal-algorithm. Prerequisites to this process are:

- Installation and configuration of the AWS CLI
- Creation of ECR repositories "multimodal-inference" and "multimodal-training" in the us-east-1 region
- Docker CLI

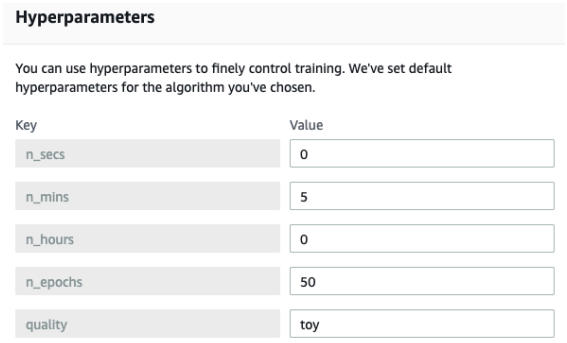
If all of these prerequisites have already been met, then building of the algorithm will be possible through the following steps:

1. First build the "base" images, by running "tRAIN_ALG-BASE/build_base_images.sh" and "iNF_ALG-BASE/build_base_images.sh" respectively. The main training and inferencing containers will build off of these "base" images.
2. Build the inferencing and training containers by running "aLG-IMGS/INF_BUILD_PUSH.sh [tag] [reg]" and "aLG-IMGS/TRAIN_BUILD_PUSH.sh

[tag] [reg]", where [tag] can be whatever you want (usually a version name for the algorithm - e.g. "latest", "alpha", "beta", etc) and [reg] is the 12-digit AWS account ID on which the ECR containers "multimodal-training" and "multimodal-inference" have been created.

3. Using the SageMaker Algorithms feature, create an algorithm using the URLs of the ECR repositories for the training and inferencing algorithms and the tags used when pushing to those repositories.
4. Follow the algorithm creation menu until the "hyperparameter specifications" page. Here, paste in the contents of "hyperparameter_specification.json". This will determine the hyperparameters that the algorithm will support.
5. On the "validation specifications" page, select no for both options and create the algorithm.

Once these steps have been completed, the algorithm will be ready to perform a training job. Before starting a training job, the following default hyperparameters are available to be changed:




The "toy" quality should not be changed for this version of the project, unless manual inferencing through the use of a Jupyter notebook is preferred. The reasons for this are flushed out in the "shortcomings" section. In the case that manual inferencing *is* preferred, other options for the quality hyperparameter are "default", "light", and "very_light". Ensure that the train data matches the header requirements described in the algorithm summary. After a training job on the algorithm completes, a batch transform job can be run (currently, only batch transform jobs are supported – hosted inferences are not). If the data provided for batch inferencing is greater than 5MB, select "Line" as the split type for that data (no other split types are currently supported).

# Shortcomings and Future Improvements

## For this Project

*Lack of Distributed Training*

Although distributed batch transforms are supported by design by SageMaker inferencing containers, distributed training has not been implemented, so training is less feasible at scale. A major improvement to the scalability of the project is distributed training support.

*High Training Memory Usage*

Due to the implementation of the algorithm, all data must be read into memory. A possible further improvement upon this aspect of the project is the implementation of support for the SageMaker training from "pipe" channel, and a way for data to be trained upon in batches. Another issue is that the nature of the algorithm demands imputation of all columns – this is especially memory-consuming for images, as the current implementation stores a bunch of 0s for each imputed image. An improvement upon the imputation steps of the algorithm would help here.

*Inferencing Container Breakage at High Model Size*

At high model sizes (model artifact output > 1GB), the inferencing container fails to respond to pings with a 200 OK response. However, since the same behavior occurs when a dummy inferencing container which only responds to pings with 200 OK is used, it is unclear whether or not this is a problem with the container, or a problem with SageMaker. For this reason, however, the quality hyperparameter must be set to "toy" unless manual inferencing will be done, as higher-quality models ended up resulting in model sizes which were too large for the inferencing container to start.

## For SageMaker

*SageMaker Documentation*

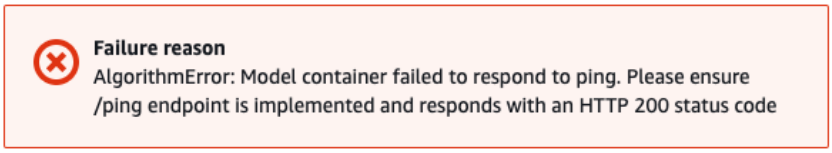
With respect to building training and inferencing containers, relatively little documentation exists. What documentation there is (https://docs.aws.amazon.com/sagemaker/latest/dg/your-algorithms.html) speaks highly generally to the requirements of the containers, and in most cases points to a GitHub repository for SageMaker containers (https://github.com/awslabs/amazon-sagemaker-examples). Although digging into the provided Jupyter notebooks on such repositories provides some more information about how SageMaker uses and expects containers, the provided information is in some cases inaccurate or hard to understand[1]. Also, there is no way provided for developers to test these notebooks without having an AWS account and spending lots of time and money testing training jobs which will fail and combing through the associated CloudWatch logs. A rewrite and expansion of these notebooks, as well as an integration of these notebooks' information to AWS documentation, would go a long way in

helping developers. Also, tools should be provided for mocking the processes which occur during SageMaker algorithm testing (the algorithm validation step, for example), so that developers can easily create algorithms without spending large amounts of time and money doing so.

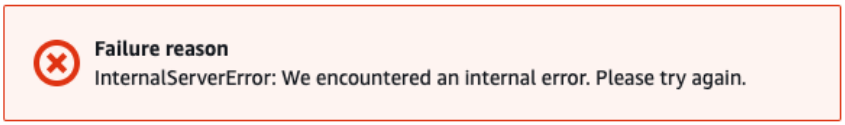
[1]Example: This notebook (last updated ~6 months ago), states the existence of a file /opt/ml/input/config/hyperparameters.json. However, it has been my experience that this file does not exist.

*Breakage at High Model Artifact Sizes*

At model artifact output sizes that were over 400MB (corresponding to models with sizes of ~100-200GB), SageMaker consistently gives the following error message:




Furthermore, at very large model artifact sizes of 5GB or higher (corresponding to models with sizes of ~1TB), the service also occasionally gives the alternate error message:




In both cases, no CloudWatch logs are generated, and there was no feasible way to dig any deeper and to debug the reasons for these errors (without rebuilding an inferencing container that replicated the AWS GitHub examples but with different technologies from scratch, which was not doable in the given time frame). As both errors occurred using even a dummified version of the code provided by awslabs' own GitHub examples, there are only 2 possible explanations:

1. In the best-case scenario, the code and base containers provided by the AWS examples are not scalable or are in some way defective for this use case.
2. In the worst-case (and somewhat more probable) scenario, the SageMaker service itself is incapable of running batch transform jobs with high model artifact sizes. Given that the SageMaker documentation regarding batch transform jobs makes no mention about of limits on the model artifact files, this is simply a bug with SageMaker.

A thorough investigation into the reasons for these errors is likely necessary to improve the SageMaker service for use cases which generate very large model artifacts.

*Unclear Data Split Options*

For SageMaker batch inferences, since the inferencing container is essentially a web server responding to inferencing requests, any data larger than 5MB must be split into smaller parts, so

that each request's payload does not exceed the web server's max payload size. SageMaker has provided a solution to this problem through the batch transform "Split type" option. The easiest split type to use is the "Line" split type, which splits the data by lines, and this option works well for use cases including CSV files which require 1 inference per line. However, this assumes that the input CSV has rows which fit in a single line. A problem during the development of this project was the use of a dirty CSV file with rows that contained newline characters. Although this CSV file was perfectly valid and was read correctly by the Python pandas library, SageMaker split the data by newline character. Although the naming of the "Line" split type heavily implies that it would do so, it nevertheless would be more user-friendly to implement a similar split type which would instead split by "Row", or CSV rows, as opposed newline characters.

# Scripts and Links

**Jupyter Notebook to make manual inferences given a training job model artifact:**

https://github.com/Derposoft/multimodal-algorithm/blob/master/CONTAINER%20inference.ipynb

**Python script used for image processing:**

```
from PIL import Image
from io import BytesIO
import base64
img_byte = BytesIO(base64.b64decode(df_image['img_jpeg'][0]))
im = Image.open(img_byte)
display(im)
```