# Content Classification using Machine Learning

Balagovind G
Mohammed Irfan
Noufal Ibrahim
Asif ETV

# Automatic content moderation using Machine Learning

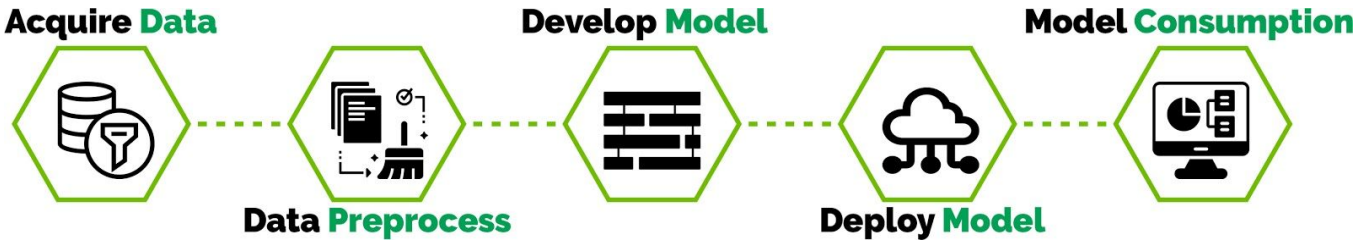
## Introduction

Our client required a machine learning solution to automatically identify bad actors from content which they input online. This was traditionally done using manual moderation which they wanted to move away from. Hamon helped them achieve their goals through our expertise in developing and deploying machine learning models.
Two of the problems the company wanted us to address through AI/ML techniques were

1. Classifying the text information that users provide into multiple categories
2. Deploying the ML model(s) to a remote server and providing APIs for classification

In this paper, we discuss in detail the technical aspects of the engineering process from Data scrubbing to Final deployment of the solution.

## Hamon's Framework for Machine Learning




Hamon Technologies has developed a standard framework for solving complex AI problems that can be replicated across most of the machine learning projects we undertake. Clients come to us with various needs and based on their situation we may choose to use some or all of the steps in the framework.

In this specific case, the company had already acquired data for the last few years. So the company could provide this to us for developing the model. Similarly, the models were exposed to clients using an API because of which the Model consumption was a simpler project.

# Classification Solution

In order to tackle the aforementioned issues in classifying text, Machine Learning algorithms were implemented in order to understand the context of the texts. Before the implementation of the ML classifiers, we preprocessed the data to avoid repetition and those which would hinder the classifications.

The technologies used in the design of the solution are:

- Preprocessing techniques in Sklearn
- Machine Learning using Python and Sklearn
- Docker & docker-compose
- Flask web framework
- SQLAlchemy
- Postgres database
- Nginx HTTP server

# The classification system

## Data PreProcess

In order to help the model with classification, the data needs to be preprocessed. The main aim of preprocessing any data is in order to reinforce featurization. In the context of machine learning a feature of a data is a numerical value which describes any property of that data. If preprocessing is done correctly, it increases the predictive power of machine learning algorithms by creating features from raw data that help facilitate the machine learning process.

Substituting parts of the data which falls into a single category, such as contact information, with a single string is one of the methods we implemented to clean up the available data.

Few of the challenges we faced in data preprocessing step includes removing website links and malicious code and substituting them with strings more appropriate and easier to be 'featurized'. Part of preprocessing also included changing the labels for the appropriate models. For example, the first model need only two labels such as , 'A' and 'B' while the second model required sub classification for the 'B' data into labels such as ‘Q1’, ‘Q2’ ‘Q3’, ‘Q4’ and ‘Q5’ to specify the nature of the 'B' data. Thus two different kinds of preprocessing were developed, one for each model.

## Develop Model

Initially, the model was designed to classify the texts into a total of 6 classes ($A_1$, $A_2$, $A_3$, $A_4$, $A_5$, $A_6$) on all the categories possible. We trained 4 models with different classifiers in order to compare the accuracy.

- Logistic Regression Classifier
- Stochastic Gradient Descent Classifier
- Bernoulli Classifier
- Naive Bayes Classifier

### Pipeline

Post preprocessing, the data still needs to be transformed before it can be fed into the classifier for training. This needs to be done for all the data that is being fed to the classifier. The purpose of the pipeline is to assemble several steps that can be validated together.

For this, we created a pipeline which comprises of three steps, two transforms and a final estimator which is the classifier. The estimator would be the classifier of choice.

### Classifier Tuning

After tuning the classifiers to output optimal accuracies, our initial accuracy level was below 70%. Logistic regression was found to be the most accurate. We then grouped the classes into 'A' and 'B', with the latter having the rest of the five subclasses. This approach required two different models; the first classifies the data into 'A' or ‘B’ and if it is the latter, the second model classifies the data into one of the five 'B' categories, ‘B1’, ‘B2’, ‘B3’, ‘B4’ and ‘B5’.

The first model straight away increased the  cross-validation accuracy to above 75%, while using the Logistic Regression Classifier. We saw that the rest of the classifiers still showed relatively poor accuracies so we rejected them.

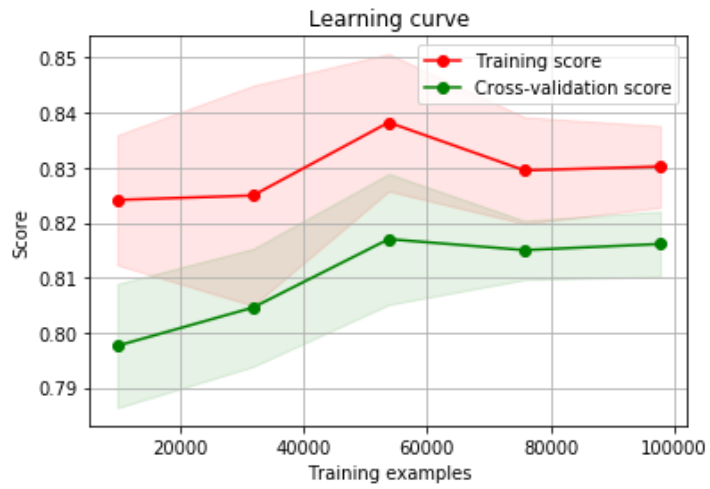
To clearly identify where we needed to tune the classifiers, we plotted a confusion matrix of the models to check the sensitivity and specificity. With that it became clear that while we almost never misclassified a 'A' as a 'B', many of the 'B' classes where being misclassified as 'A'. This was problematic since the client application has a very low tolerance for 'B's classified as 'A's.

The model was biassed in favour of 'A' (based on prior distribution) and the sample had to, in a sense, prove that it was 'B' using features. We changed that to shift the priors towards 'B'. In an abstract sense, we made the classifiers consider samples to be 'B' unless proven otherwise.

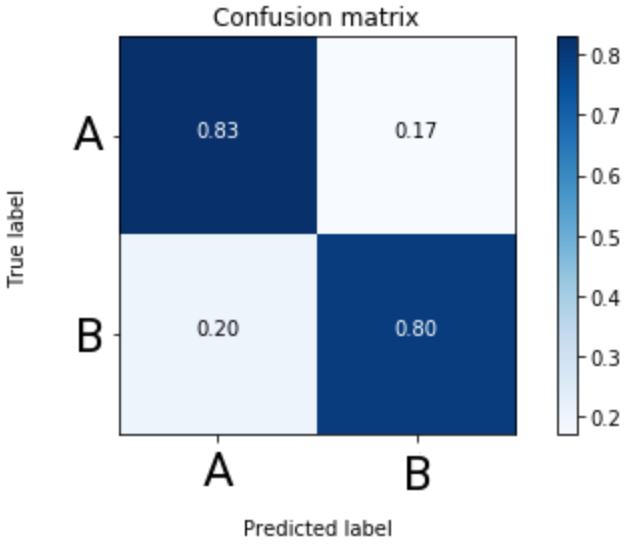
Adjusting the class weights in favour of 'B' class solved the issue. Cross-validation accuracy increased to 80% and above post that tweak.

One problem Hamon noticed regarding the data, as we mentioned in the downside of doing the classification manually, is that there were many ambiguous classifications. We found that Many of the 'A's were classified to one of the categories in B's (B5)s. This error in the training data caused the machine to be confused between both the classes. Hamon brought this to client's attention and educated them on the importance of being careful about the data which is used to train the model.

Hypertuning the models improved the accuracy and the confusion matrix by a couple of percentages making the cross-validation accuracy to over 80%. Plotting the Learning Curves told us how much data was enough to get the job accurately done.

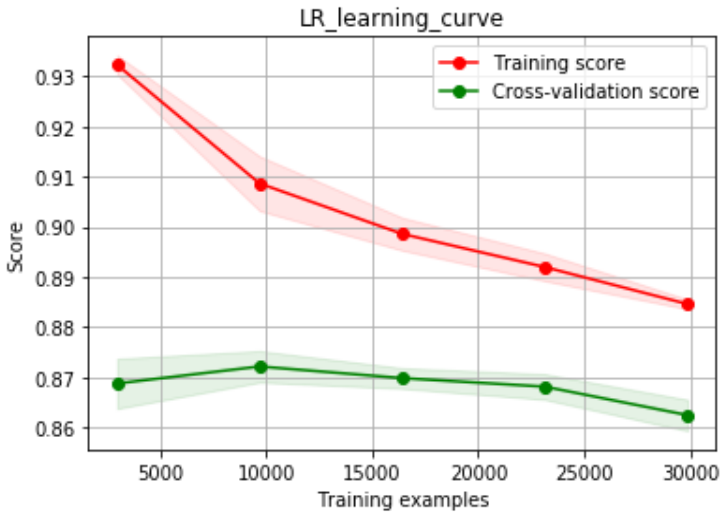



**Learning Curve of Model 1**

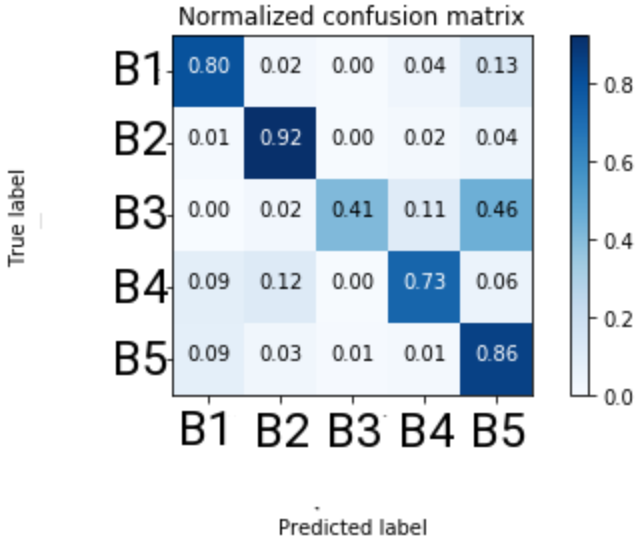



**Confusion Matrix of Model 1**

The challenge with the second model which classifies class 'B' into 5 subclasses was the irregularity in the composition of different classes in the dataset. This caused the confusion matrix to be in favour of one of the subclasses, '$B_5$', which was higher in number. Here we took a different approach to normalize the dataset by taking only fractions of the classes which are higher in number such that finally the data available on each class is balanced. This gave us a much better confusion matrix and a cross-validation score of above 85%.




**Learning Curve of Model 2**




**Confusion Matrix of Model 2**

## Retraining the Model

One of the tougher problems we had to solve was the retraining of the model with batches of data in order to improve the accuracy. The confusion matrix was wrong and the accuracy could not be trusted because of the smaller number of data provided for retraining. One thing we noticed is that the number of features was different each time the pipeline was 'fit' with new data because of which the 'warm start' parameter used to fail. Thus we pulled the Classifier out of the Pipeline. Now the pipeline comprised only of the

transformers. This was to make sure that the feature count remained the same even despite training the classifier repeatedly.

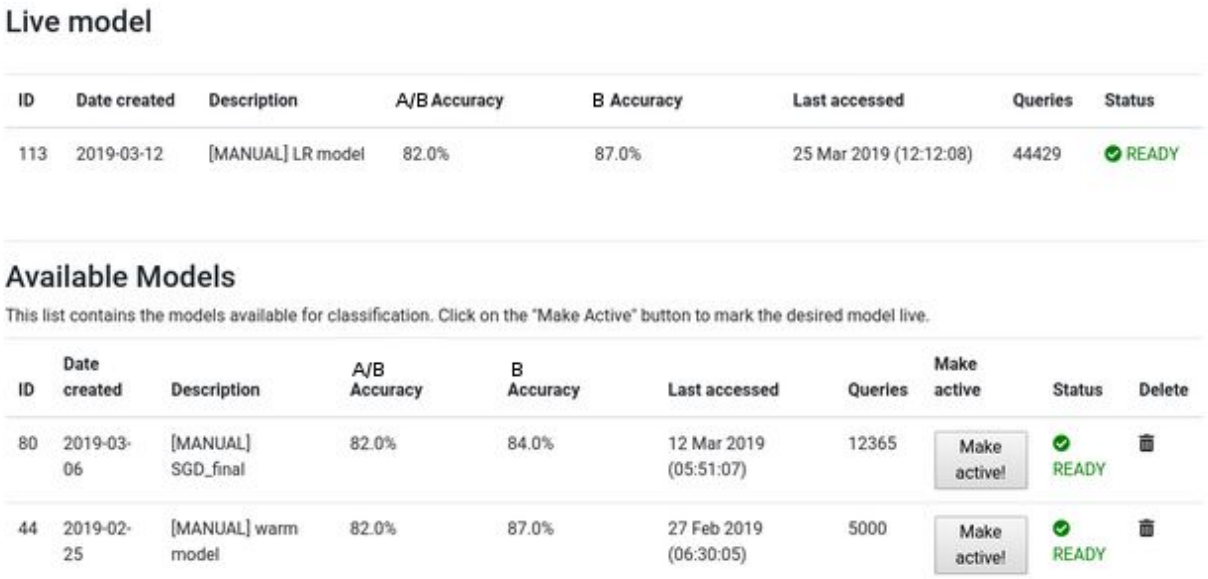
When the retraining issue still persisted, we tackled it the hard way possible. The original dataset was fairly big and the re-training dataset was comparatively small. So we substituted the data of each class in the former dataset with the new re-training data such that the total dataset still remained the same size. This way the class weights and the data size, the model is being trained with, remained the same.

Just as we suspected, the cross-validation accuracy and the confusion matrix returned the same, if not improved at some places, for the second model. While it declined a little for the first model, which we noticed was mainly because of the discrepancy of the input dataset where many of the 'A's were mislabelled as '$B_5$'s. Except for that, the retraining worked well.

## Deploying Model

Arguably the most important part of the project was to expose the classification engine in a usable way. A REST API was developed in Flask to provide easy access to the classification results. A single endpoint would receive either a GET or POST based text query, run it over the set of pre-trained models and return the class it belongs to, the subclasses (if applicable) and the predicted language and their accuracies.

A dashboard web page was built to manage the set of models which also included its characteristics such as accuracy, ID, date of creation, number of queries etc. The end-user could select the IDs of the model to use when making queries or rely on the default 'live' model that can also be set from the dashboard.



## Live model

| ID | Date created | Description | A/B Accuracy | B Accuracy | Last accessed | Queries | Status |
|---|---|---|---|---|---|---|---|
| 113 | 2019-03-12 | [MANUAL] LR model | 82.0% | 87.0% | 25 Mar 2019 (12:12:08) | 44429 | READY |

## Available Models

This list contains the models available for classification. Click on the "Make Active" button to mark the desired model live.

| ID | Date created | Description | A/B Accuracy | B Accuracy | Last accessed | Queries | Make active | Status | Delete |
|---|---|---|---|---|---|---|---|---|---|
| 80 | 2019-03-06 | [MANUAL] SGD_final | 82.0% | 84.0% | 12 Mar 2019 (05:51:07) | 12365 | Make active! | READY | |
| 44 | 2019-02-25 | [MANUAL] warm model | 82.0% | 87.0% | 27 Feb 2019 (06:30:05) | 5000 | Make active! | READY | |

**Dashboard**

To prevent unauthorized usage of the service, all of the endpoints were protected with the standardized basic access authentication over https which is essentially a username and password based authentication that is also natively supported by many user-agents such as all modern web browsers.

Newer models can be generated by retraining an existing model with new data from an excel sheet. For this, we included an excel upload feature on the dashboard to retrain the current live model on the database. Since the retraining process was time-consuming, it had to be done inside a thread at the backend side. While this worked initially, we identified several problems with this flow. Being a long running task, there's a possibility that an error or exception at any point could be overlooked. Even if we could log this onto the dashboard, we found it more sensible (and quicker) to implement a command line interface just for this. Command line interfaces inherently give us more visibility over the errors (and the resulting stack traces) which can be very useful for debugging.

Models used in this web service were stored as a serialized version of the pipeline object (which is essentially a pre-trained model instance) in the database and each of this consumed around 50-60 Megabytes. Accessing and deserializing large binary objects from the database per request is a time-consuming process. We had to rely on caching at the application level

to keep some of these in memory. Since most of the deserialized models are now kept in-memory, the WSGI server (gunicorn) hosting the flask processes began consuming substantial amounts of memory. Gunicorn being a pre-fork worker model would essentially keep duplicated copies of the memory intensive flask application at runtime (even before the requests came in). Thus even with just a few models and little or no traffic, the average server with 4GB of RAM had a tough time with out-of-memory killers. To overcome this, we configured gunicorn to run the workers as threads instead of processes. Since memory is shared between threads, we now have a reduced memory footprint at the server side. This change, combined with increased swap space, a tweaked worker thread count and optimized SQL code, we were able to fully run the system (and all operations that required a bit of memory like inserting the models into the database, retraining etc) without any issues.

The entire system was deployed onto a single 4GB RAM VPS. This included the flask application running over gunicorn along with a Postgres database. All of this was running inside Docker containers that made it easier to deploy the service onto the machine. Docker containers enforce a particular version of the python runtime and every other dependency that our service has. So platform level inconsistencies are no longer a problem. A docker compose file helped us to run all of this in one single command. Next, we created a systemd service and configured it to start our docker containers using docker-compose. The only application running outside of the container was the Nginx HTTP server that acted as a reverse proxy to the gunicorn/flask application. The end result was a very portable and resilient deployment solution for our project.

# Conclusion

We developed and deployed an efficient easy-to-integrate classification engine with retrainable models that gave us good prediction outcomes. With an initial accuracy of over 80%, we were able to place our client in a position to reduce the number of human moderators over time and improve upon their existing content classification methods.