GSA AI Challenge - Description of Methods

[**Overview of Submission**](#_aoroktxlg0e) **2**

[System Diagram](#_25ttda8h5eqr) 3

[BEULAR - Web Application](#_zcg59gh6iaz8) 3

[BEULAR-API - REST API and AWS Cloud Resources](#_wxb3vlthl2uj) 4

[BEULAR-NB - Modeling Source Code](#_hcygxr8nzknz) 4

[**Description of Data**](#_qgqs3zpzz5du) **4**

[**Exploratory Data Analysis**](#_xaynm62xl65o) **5**

[Tools Used](#_lxag5fol00hm) 5

[Findings](#_u4ephgpmiy9d) 5

[Data Duplication](#_a1eslhvrdnda) 5

[Class Imbalance](#_7my2ta75829y) 5

[Data-Generation Process Concerns](#_jf8ff718j5qf) 6

[Clause Definition](#_h93mbk99egbg) 6

[Training Data Creation](#_uuofba8wftom) 6

[EDA Summary](#_q7cmr2hdeua) 6

[**Modeling Methods**](#_37v4s96z7pkb) **7**

[Train-Test Split & Cross Validation](#_fr8a8gc1iomf) 7

[Model Pipeline](#_2ke8eox893vg) 9

[Feature Generation](#_un65xkmg7ta) 9

[TF-IDF](#_amhjn0ic0xh2) 10

[Lemmatization](#_7jp7tenokt41) 10

[Stemming](#_28w6uthkr8bt) 10

[Default Tokenization](#_ejt8xrz3j90p) 10

[Word Embeddings](#_rkjxlem0w2wi) 10

[Feature Selection](#_jkxn9s46g5wm) 10

[Estimators](#_qa7kam2pnld7) 11

[Linear Models](#_2k02jzgxnf0) 11

[Logistic Regression](#_te715xbllrj9) 11

[Support Vector Machine](#_mf2kaudilnux) 11

[Word Embedding Model](#_ucu1cpcx8k3i) 11

[Blazing Text](#_ot92qekmkib3) 11

[Decision Tree Models](#_1bji8dpi207e) 12

[Random Forests](#_q9rmpzfvar1m) 12

[Gradient Boosting](#_7yen195rnbt2) 12

[Model Selection](#_lorvs4s9ync8) 12

[Model Performance](#_awc2g05qd7fz) 13

[**BEULAR - Web Application**](#_4yx3tioltz7t) **13**

[Framework](#_v7zjnc1rpfc) 14

[Dependency Management](#_owxgfhddh756) 14

[Web Framework](#_3omey5y8fib9) 14

[Service Worker](#_l59g3ta1804q) 14

[Text Extraction](#_161fqd79ihd4) 14

[Hosting](#_ljik6v4yth43) 14

[User Experience](#_gwvf3ncetjib) 14

[**BEULAR-API AWS Cloud Backend**](#_kk1nt3cs4dq3) **15**

[Infrastructure as Code](#_1w2vli37bnze) 15

[Model Building, Training and Deployment](#_73hcn6w0ymwh) 15

[Data Storage](#_jhaebc2q4nxs) 15

[Model Building](#_qssyrbrry48g) 15

[Model Training](#_sr4kbd3alohe) 15

[Model Deployment](#_enaqtik011ne) 15

[REST API](#_be06e0wg95ze) 16

[**Next Steps**](#_t08c6pk9nwu2) **16**

[Data Cleansing](#_vrcq76ypiwcl) 16

[Use the Validation Data](#_z0v65oilbkgo) 16

[API Authentication and Authorization](#_gtoz7e2m1v9h) 16

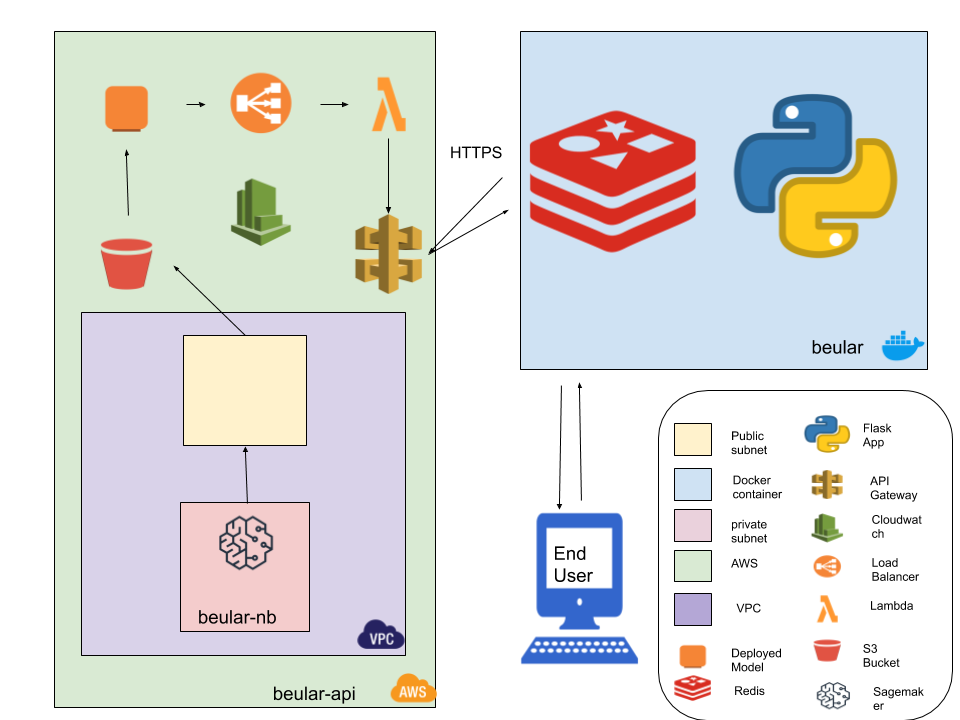
[Auto-scaling Group for the REST API](#_938x8347brts) 16

# Overview of Submission

Binarization of End User License Agreement Recommendations (BEULAR) consists of three separate source code repositories. Together, they create a dockerized web application that integrates with an AWS backend for data storage and model training, deployment and hosting via a REST API. In order to test the app without having provisioned the prerequisite AWS sources, the web application component can optionally run offline.

## System Diagram

This web-app position of this submission is designed to work locally via Docker with the back-end optionally hosted in AWS. If AWS is not used for a back-end, the model(s) can be included in the Docker container (see web application’s README).



## BEULAR - Web Application

This repository contains a dockerized web application. The application allows users to upload one or more documents and review the model’s predictions on each clause in the document(s). Each prediction is explained with details on the classifier’s probability score and the contribution of each word/phrase to the model’s decision.

If the application is linked to an AWS account that has provisioned the AWS resources created in the other two repositories, the application requests clause predictions from a REST API and uses a Redis worker to capture and save user feedback on each prediction in order to re-train the model with more data. Without the AWS integration, clause predictions are provided by a model that is packaged with the app itself and user feedback is not saved.

Clauses are extracted from uploaded documents with Python’s textract library, which uses underlying system libraries that are a part of the base Docker image. Although AWS (and other cloud providers) have text extraction services, these are expensive and likely overkill given the high quality of sample documents provided by the government. Moreover, the textract library supports many document formats, including doc, docx, rtf, txt, and even powerpoint and excel.

## BEULAR-API - REST API and AWS Cloud Resources

This repository contains an AWS Cloud Developer Kit (CDK) application that uses infrastructure-as-code to provision the following AWS resources:

1. Virtual Private Cloud - network infrastructure
   1. Public subnet
   2. Private subnet
      1. NAT Gateway and Internet Gateway used to provide egress-only internet access
2. SageMaker - machine learning service, placed within the private subnet of the virtual private cloud instance, with the internet disabled for the sagemaker instance so as to inherent network configuration of the private subnet
3. S3 bucket - non-public bucket for a data store
4. Lambda - serverless proxy functions for handling REST API requests to hosted models
5. API Gateway - REST API for the deployed SageMaker models

The Sagemaker, VPC and S3 resources were used for model training. The Lambda and API Gateway resources create a REST API for a model that has been trained and deployed using SageMaker. Creating this REST API is optional, as the trained models have been downloaded from the S3 bucket and packaged with the web application. The government can use environment variables to configure the application in this regard (the README in the repository explains in depth).

## BEULAR-NB - Modeling Source Code

This repository contains the source code for the exploratory data analysis, modeling, and post-hoc analysis that I performed. This repo is meant to be linked to the BEULAR-API repository so that its source code can be git pulled/pushed from the SageMaker instance. Any notebook that imports the sagemaker Python module is meant to run within AWS Sagemaker.

# Description of Data

The training data has been extracted from actual EULA documents and had identifying information, such as company name, removed. A known issue was that clause text contained control characters, such as embedded New Line (/n) characters.

The training data is in csv format with the following columns:

|  |  |  |
| --- | --- | --- |
| Clause ID | Clause Text | Classification |
| Integer generated for tracking individual clauses. | Section or paragraph of a EULA document that has been reviewed for acceptability to GSA. | Section or paragraph of a EULA document that has been reviewed for acceptability to GSA. |

# Exploratory Data Analysis

## Tools Used

Most data analysis and modeling was performed in Python using the Amazon Web Services (AWS) SageMaker service. The AWS Cloud Developer Kit (CDK) was used to write the infrastructure-as-code, which will allow the government to quickly and easily replicate the environment and resources that I used. This process is documented in the beular-api repository.

Within Python, I used the pandas, numpy, sklearn, matplotlib and nltk libraries.

## Findings

### Data Duplication

526 samples are duplicates. These were dropped from the dataset prior to training. Not doing so would artificially inflate performance whenever a duplicate was present in both the train and test datasets. Two of the samples (clause #5250 and #5249) have different labels despite having the same text. These were also dropped.

### Class Imbalance

Approximately 22% of the training data was a positive sample. This imbalance likely reflects reality, as the majority of clauses in a EULA are not positive samples. Nevertheless, class imbalance makes classification more challenging, often requiring the presence of more training data. In the absence of more training data, a variety of under- and/or over-sampling techniques can be applied to create a more balanced dataset. I applied the Synthetic Minority Oversampling Technique (SMOTE)[[1]](#footnote-0) in some experiments, but the results did not improve. This is likely due to the issues identified in the next section.

### Data-Generation Process Concerns

The first concern relates to how a clause is defined. The second concern relates to the lack of instruction for using the training data.

#### Clause Definition

No concrete definition for what constitutes a clause was provided, so the structure of the training and validation data files is the only cue. In the training data, 3% of the clauses were just a single word (and none of these were positive samples); and nearly 10% were three words or less, with only 4 positive samples in that bunch. A particularly perplexing example is the clause “company confidential”. Another is the single-word clause “DEFINITIONS”. I’m not sure how these constitute a clause.

After calculating the number of words in each clause, I performed a one-way t-test to determine that there is a statistically significant difference between the mean number of words for positive and negative samples (see the EDA Jupyter Notebook). This suggests that the training data contains too many “clauses” of just a few words that do not in fact constitute a “clause”.

#### Training Data Creation

Typically, training data for machine-learning competitions is already split into training data and test data, with both sets containing class labels, and a separate validation dataset whose class labels are hidden from competitors so that it can be used for judging submissions. This wasn’t the case for this competition.

For data scientists, the test dataset provides a stand-in for the trained model’s ability to generalize to the validation data after training on the train data. Since the competition did not perform this train-test split for contestants - or provide them with a random seed to use when splitting themselves - competitors’ models have no fair basis for being judged relative to one another.

Although I set a random seed to ensure the reproducibility of my results, one should be wary of making comparisons between contestants’ models’ performances since differences could be attributable to the randomness of the train-test split. This is especially true in the presence of such a small training dataset with an imbalanced distribution of positive samples.

## EDA Summary

Due to these data-generation concerns, I am doubtful about the ability of probabilistic models to accurately learn and, more importantly, generalize to unseen data. Thus the model I used to generate predictions for the validation data did not perform a train-test split and instead relied upon cross-validation during the hyperparameter search in order to ensure generalizability to the validation data. This method is explained in greater detail later, but it essentially performs multiple random train-test splits so as to mitigate the influence of manual data splitting.

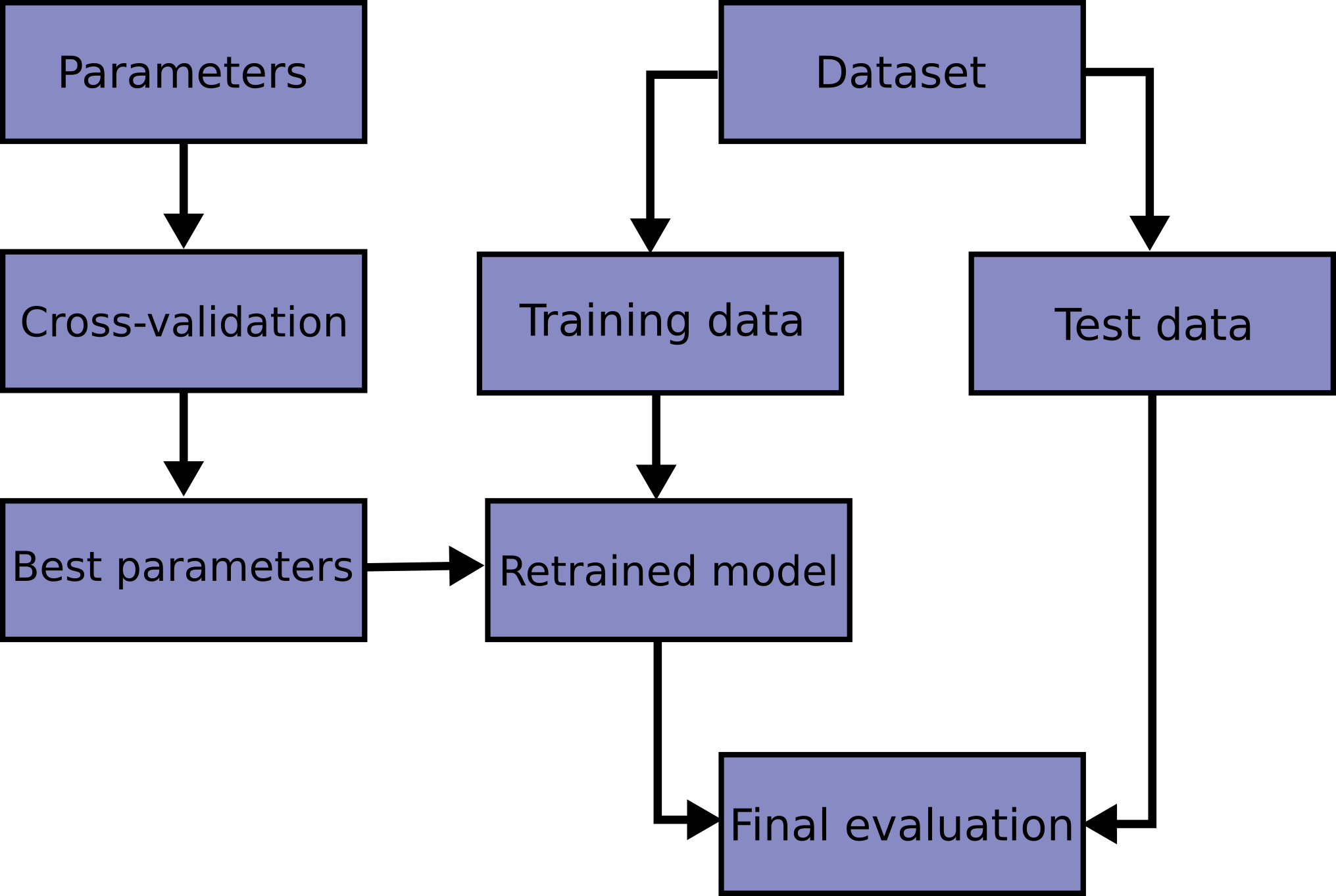
# Modeling Methods

The goal of predictive modeling is to develop a model that makes accurate predictions on new data, unseen during training. This is a hard problem because we cannot evaluate the model on something we don’t have. Therefore, we must estimate the performance of the model on unseen data by training it on only some of the data we have and evaluating it on the rest of the data. This is the principle underlying the reason to split data into train and evaluation sets as well as more sophisticated techniques, such as cross validation.

## Train-Test Split & Cross Validation

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called *overfitting*. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set.

Since this competition did not provide a random seed for splitting the training data, nor provide guidance on how much data should be used for a test set, I chose to split the data 80-20. During training with the sklearn models, I also used a process called k-fold Cross Validation (CV) to further mitigate overfitting. The blazingtext model samples data in batches, which amounts to the same thing. Thus the data flow during training looks like this:



With k-fold CV, the training set is split into k smaller sets. I used a value of 5 for k, which results in five random 80-20% splits of the data during each iteration of the hyper-paramter search. During the search, the following procedure is followed for each of the k “folds”:

* A model is trained using k-1 of the folds as training data;
* The resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by k-fold cross-validation is then the average of the values computed in the loop. The following image illustrates the process:

## Model Pipeline

The modeling process is a pipeline of transformations. The first step is feature-generation, followed by an optional feature selection algorithm, and ends with an estimator, which is what returns class predictions and their respective probabilities.

### Feature Generation

Two methods were used to generate features from the corpus:

1. Term-Frequency Inverse-Document Frequency Vectorization
   1. Lemmatization Tokenization
   2. Stemming Tokenization
   3. Default Tokenization
2. Word Embeddings via transfer learning from the BlazingText implementation of the Word2Vec algorithm[[2]](#footnote-1)

#### TF-IDF

In information retrieval, TFI-DF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection of documents. It is often used as a weighting factor in searches of information retrieval, text mining, and user modeling. The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the collection of documents that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

##### Lemmatization

Lemmatization is the process of grouping together the inflected forms of a word so they can be analysed as a single item. For lemmatization, I used Python’s NLTK library, applying transfer learning from the pretrained wordnet model[[3]](#footnote-2).

##### Stemming

Stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form. For stemming, I used the Porter algorithm in Python’s NLTK package. For example, stemming reduces, argue, argued, argues, and arguing to the stem “argu”.

##### Default Tokenization

The default tokenization method used a regular expression that selects tokens of 2 or more alphanumeric characters.

#### Word Embeddings

The Word2vec algorithm underlying the BlazingText model uses transfer learning to map words to high-quality distributed vectors. The resulting vector representation of a word is called a word embedding. Words that are semantically similar correspond to vectors that are close together. That way, word embeddings capture the semantic relationships between words.

### Feature Selection

Feature selection reduces the number of variables going into a model and thereby mitigates the curse of dimensionality, which manifests primarily as increased computation time and decreased generalizability.

To select the optimal features from a sparse feature matrix, I used a Truncated Singular Value Decomposition. Using this algorithm on a tf-idf matrix is commonly known as Latent Semantic Analysis. For the BlazingText algorithm, which is a neural network, a black-box hidden layer performs this step.

### Estimators

When performing classification, you often want not only to predict the class label, but also obtain a probability of the respective label. This probability gives you some kind of confidence on the prediction. Some models can give you poor estimates of the class probabilities and some even do not support probability prediction. Since the competition required using a probabilistic classifier, the following estimators were used for experimentation:

#### Linear Models

##### Logistic Regression

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable.

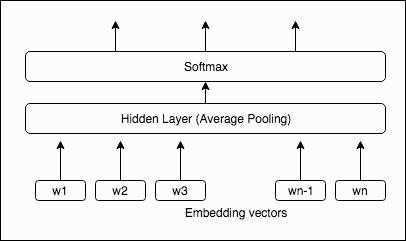
##### Support Vector Machine

A Support Vector Machine model represents samples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall. Platt scaling is used to make the model probabilistic.

#### Word Embedding Model

##### Blazing Text

BlazingText is Amazon’s GPU-accelerated implementation of the fastText text classification model, which utilizes transfer learning to generate word embeddings[[4]](#footnote-3) for each token in a clause. This results in a vector representation of each token, which is aggregated with average pooling before being passed to a softmax function that outputs a class probability as a prediction.



To generate the word embedding vectors, a lookup is done on an embedding matrix generated by days of unsupervised machine learning on a large corpus of text. These word representations are then averaged into a text representation, which is in turn fed to a linear classifier.

Since bag-of-words is invariant to word order, the model uses bag-of-words n-grams as additional features to capture some partial information about the local word order. For instance, in the previous figure, if 2-word n-grams are used for training, then vector representations for w1w2, w2w3,…. wn-1wn will be learned and used for computing the hidden layer along with the individual word vectors.

#### Decision Tree Models

##### Random Forests

Random forests or random decision forests are an ensemble learning method for classification that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

##### Gradient Boosting

The Gradient Boosting Classifier is another ensemble model of decision trees. Gradient boosting relies on the intuition that the best possible next model, when combined with previous models, minimizes the overall prediction error. The key idea is to set the target outcomes for this next model in order to minimize the error.

### Model Selection

For all but the BlaingText estimator, learning was accomplished using a randomized grid search of hyper-parameters. The linear models utilized **stochastic gradient descent** learning whereas each of the decision tree methods implement a variation of bagging, which is a resampling technique. All training methods utilized cross-validation during training in order to prevent overfitting and poor generalization on the validation dataset

### Model Performance

The Analysis.ipynb notebook of the beular-nb repository provides a much more in-depth analysis of each model.

The following table summarizes the performance of each model on the *test* dataset:

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Accuracy** | **F1** | **Brier Score** |
| **SGDClassifier** | **.79** | **.60** | **.15** |
| RFC | .78 | .56 | .15 |
| GBC | .83 | .49 | .15 |
| BlazingText | .84 | .53 | .12 |

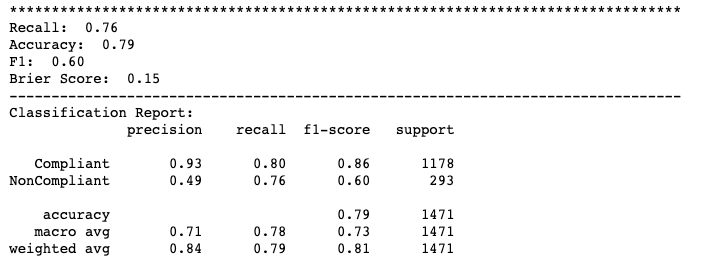
Research has found that a high degree of variance in the explanatory and noise variables results in logistic regression consistently outperforming ensemble models such as random forests and gradient boosting[[5]](#footnote-4). Conversely, the true positive rate for ensemble models tends to be higher than logistic regression in the presence of many noise variables, which I believe to be the case with this dataset due to the data-generation issues I outlined earlier.

Although gradient boosting does run the risk of overfitting (high variance), I applied several techniques to mitigate this. First, I applied regularization via shrinkage (learning\_rate < 1.0), which can improve generalization performance considerably. In combination with shrinkage, I used stochastic gradient boosting (subsample < 1.0) to further reduce the variance via bagging. I also used another strategy to reduce the variance by subsampling the features analogous to the random splits in Random Forests (via the max\_features parameter).

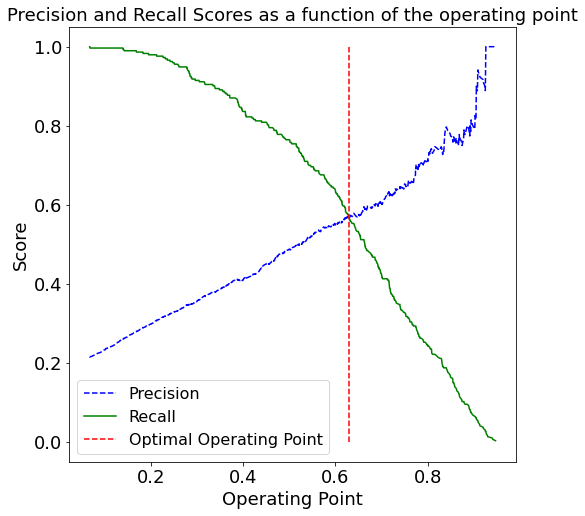
Regardless, the linear model performed best. The BlazingText model performs well on the training dataset, but it doesn’t do well in the presence of imbalance data that likely has underlying data-generation issues. Because of this, predictions for the validation data file were generated using the SGDClassifier model trained on the entire training dataset[[6]](#footnote-5).

### Best Model Analysis

Here’s a classification report for the Logistic Regression model learned with Stochastic Gradient Descent.

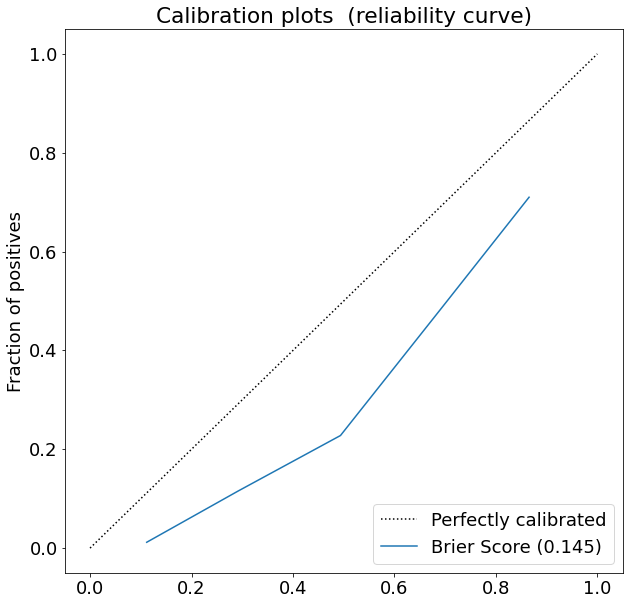


Since this is a probabilistic model, the following visuals demonstrate how it’s possible to make tradeoffs between the F1 and Brier Scores by adjusting the probability threshold of the classifiers’ prediction, which in turn affects precision and recall.

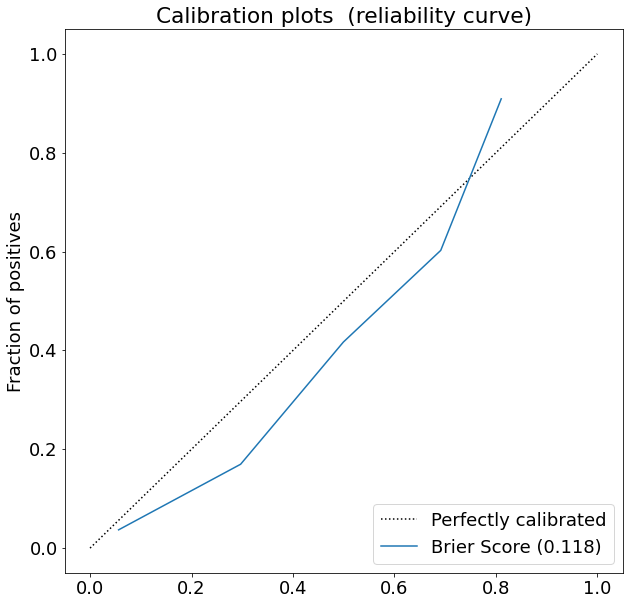


You can see that the optimal decision (operating) point, when trying to balance precision and recall, is around 62% instead of the default 50%.

When using 50% as the decision point, the model achieves a f1-score of 60% and a brier score of 15%. We can visualize that model’s calibration - which is what the brier score proxies - with this chart:



Since this lies under the diagonal, the model is predicting non-compliant too frequently. Above, we saw that our optimal decision point was about 62% If we change our decision point to 62%, we'll increase precision for the positive class (i.e. Non-compliant) at the expense of precision for the negative class (i.e. Compliant). This will, however, lead to the inverse for recall. Let's see the calibration curve if we adjust the operating point:



The model is more calibrated - i.e. lower brier score - but the f1 score has subsequently dropped just slightly to .59. This is due to the imbalanced nature of the data.

# BEULAR - Web Application

Binarization of End User License Agreement Recommendations (BEULAR) is a web application that allows business users to upload Word or pdf documents and then review model predictions on each clause from that document. Users can inspect the rationale behind predictions by seeing probability scores and the weighted contribution of each word/phrase in a clause toward the prediction. They can also validate the predictions, which are then sent to the cloud for later use in re-training and, ultimately, re-deploying a new and improved version of the model.

The application supports use of the logistic regression model, random forest model, and gradient boosting model.

## Framework

### Dependency Management

Docker is used to manage operating-system-specific dependencies and to increase the portability of the application.

### Web Framework

The application uses Python Flask as a web framework. Bootstrap is used for CSS and minimal javascript is used client side for manipulating elements on the page via the jquery library.

### Service Worker

Redis is used as a service worker to asynchronously communicate with AWS when saving feedback on model predictions. This design pattern treats the communication with the S3 data store as a microservice and prevents blocking API calls on the application’s main thread, which would increase app latency.

### Text Extraction

I used the textract python package to parse text from EULA documents. Although cloud-hosted services like AWS textract are available, they’re expensive and likely overkill for this use case.

## Hosting

For monetary reasons, the web application is not hosted anywhere. However, the app’s README documents deployment to Heroku, which would be very similar if using cloud.gov.

## User Experience

From a user’s perspective, the app:

* allows users to upload Word or PDF documents
* parses uploaded documents into their constituent clauses
* presents the predictions using color-coding to call out potentially non-compliant clauses
* allows users to see an explanation for each prediction, showing them the prediction’s probability score as well as the influence of each word/phrase in the clause toward generating that prediction
* allows users to validate predictions, with an option for providing open-ended feedback, and submits that data to a cloud-hosted datastore where the models can be re-trained and therefore incrementally improve over time

# BEULAR-API AWS Cloud Backend

## Infrastructure as Code

I used the open source AWS Cloud Development Kit (AWS CDK) to model and provision my cloud application resources using Python. Although I used Python to model the resources, it is possible to generate the Cloudformation templates in YAML format with a single command. This allows anyone to create the same cloud resources that I used.

## Model Building, Training and Deployment

### Data Storage

A non-public S3 bucket is used as a data store.

### Model Building

Models were written in Python using the sklearn library as well as AWS’s implementation of a GPU-accelerated word-embedding neural network. They can be found in the BEULAR-NB repository.

### Model Training

SageMaker was used for model training. The notebook instance is linked to the BEULAR-NB repository via git, so it can be updated easily.

### Model Deployment

SageMaker can be used to deploy models, but they are only accessible at this point via the AWS command line interface.

## REST API

To make the models available via a REST API, I used Lambda and API Gateway. Once the model is deployed, one could use curl to get a prediction from some text (if using a sklearn model):

curl -X POST -H "Content-Type: text/plain" --data "this is a test" https://fntzl3eq2h.execute-api.us-east-1.amazonaws.com/prod/

If using a deployed BlazingText model, the request would look like this in Python:

*import* requests

uri = 'https://abc123.execute-api.us-east-1.amazonaws.com/prod/'

data = {'instances': ['This is a test of the system']}

r = requests.post(uri, *json*=data)

r.json()

*#[{'prob': [1.0000100135803223], 'label': ['\_\_label\_\_0']}]*

# Next Steps

## Data Cleansing

There were a lot of duplicates and very short “clauses” in the training data. The validity of these samples should be questioned as I believe a tighter definition for clause would lend itself toward defining a better model.

## Use the Validation Data

The government could make a better model by combining the validation dataset (assuming it’s already labeled) with the training dataset and then judiciously removing 20% of it for test data. This would greatly increase the amount of training data and enable researchers to better experiment.

## API Authentication and Authorization

The REST API for the models in AWS could be enhanced with the AWS Cognito service to provide authentication and authorization for the API.

## Auto-scaling Group for the REST API

Presently, deployed models sit in a single EC2 instance. Slight configuration changes could deploy models to an EC2 auto-scaling group that sits behind a load-balancer. API Gateway and Lambda would sit in front of this load-balancer to proxy requests. A DNS service such as AWS Route 53 could also be used to give the API a recognizable name. With this change, it would also be beneficial to move model API calls to the Redis (or similar) service.

1. Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. Journal of artificial intelligence research, 16, 321-357. [↑](#footnote-ref-0)
2. Gupta, Saurabh & Khare, Vineet. (2017). BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs. 1-5. 10.1145/3146347.3146354. [↑](#footnote-ref-1)
3. Princeton University "About WordNet." WordNet. Princeton University. 2010. [↑](#footnote-ref-2)
4. Note: this model uses a different feature creation/selection pipeline than the other estimators. It relies on word embeddings instead of TF-IDF and average pooling instead of the SelectK feature selection method. [↑](#footnote-ref-3)
5. Kirasich, K., Smith, T., & Sadler, B. (2018). Random Forest vs Logistic Regression: Binary Classification for Heterogeneous Datasets. SMU Data Science Review, 1(3), 9. [↑](#footnote-ref-4)
6. I used the entire training dataset since I know the model can somewhat generalize with Cross-Validation and since it will benefit from seeing 100% instead of 80% of the training data. [↑](#footnote-ref-5)