The document details out the process of analyzing unstructured data and applying sophisticated NLP techniques such as token vectorization, word embeddings and deep learning algorithms, to predict whether a tweet denotes a real disaster event.

Disaster Tweets

Natural Language Processing

Gaurav Dembla

Nov 7, 2020

Contents

Introduction

The goal of this project is to build a machine learning model that predicts which Tweets are about real disasters and which ones aren't. Since tweets are unstructured data, we will need to apply Natural Language Processing techniques to parse information out of free texts.

Context

Twitter is a social networking service which allows users to post messages known as "tweets". While users observe or experience a disaster, be it natural or man-made, they can, and occasionally do, announce an emergency through "tweets" in real-time. However, it may not always be clear whether a person's words are indeed announcing a disaster. For example, look at the tweet – "On plus side LOOK AT THE SKY LAST NIGHT IT WAS ABLAZE". Here, the author explicitly uses the word "ABLAZE", but means it metaphorically. Though this is clear to a human right away, it is not so clear to a machine.

Source Data

The training dataset has been sourced from Kaggle. It has 5 columns and 7613 rows, which should be enough for both model training and testing on hold-out data set.

Sample records from the training dataset are as shown below. Target value of 1 denotes tweet as a disaster.

id	keyword	location	text	target
1			Our Deeds are the Reason of this # earthquake May ALLAH Forgive us all	1
4			Forest fire near La Ronge Sask. Canada	1
5			All residents asked to 'shelter in place' are being notified by officers. No other evacuation or shelter in place orders are expec	1
48	ablaze	Birmingham	@bbcmtd Wholesale Markets ablaze http:/ / t.co/ lHYXEOHY6C	1
49	ablaze	Est. September 2012 - Bristol	We always try to bring the heavy. # metal # RT http:/ / t.co/ YAo1e0xngw	0
50	ablaze	AFRICA	# AFRICANBAZE: Breaking news:Nigeria flag set ablaze in Aba. http:/ / t.co/ 2nndBGwyEi	1
112	accident	San Mateo County, CA	Traffic accident N CABRILLO HWY/ MAGELLAN AV MIR (08/ 06/ 15 11:03:58)	1
119	accident		Can wait to see how pissed Donnie is when I tell him I was in ANOTHER accident??	0

Success Criteria

ROC-AUC (area under the Receiver Operating Characteristic curve) on unseen data (hold-out set) shall be used to assess and compare alternatively trained statistical models with each other. Please refer to **Glossary** section to gain intuition behind the classification metric.

Exploratory Data Analysis

Data Balance

The data seems to be nearly balanced as % of disaster and non-disaster tweets are nearly equally distributed. We will go ahead with the data as-is and shall not apply any data balancing techniques.

Target	% of records
0	57.0%
1	43.0%

Missing Values

Based on the following figure that shows proportion of nulls in the data for each attribute, we will not consider *Location* attribute for building the model.

Attribute	% Nulls
Keyword	0.8%
Location	33.2%
Text	0.0%
Target	0.0%

After looking into the values of *Keyword* attribute, it is clear that the underlying information is redundant and that we can drop off the attribute, since the token generation algorithms themselves will generate a set of essential keywords (tokens) for model training.

id	keyword	location	text	target
1			Our Deeds are the Reason of this # earthquake May ALLAH Forgive us all	1
4			Forest fire near La Ronge Sask. Canada	1
40			Cooool :)	0
48	ablaze	Birmingham	@bbcmtd Wholesale Markets ablaze http:/ / t.co/ lHYXEOHY6C	1
49	ablaze	Est. September 2012 - Bristol	We always try to bring the heavy. # metal # RT http:/ / t.co/ YAo1e0xngw	0
50	ablaze	AFRICA	# AFRICANBAZE: Breaking news:Nigeria flag set ablaze in Aba. http:/ / t.co/ 2nndBGwyEi	1
112	accident	San Mateo County, CA	Traffic accident N CABRILLO HWY/ MAGELLAN AV MIR (08/ 06/ 15 11:03:58)	1
119	accident		Can wait to see how pissed Donnie is when I tell him I was in ANOTHER accident??	0

While looking at few tweets, I also realized that tweets are full of URLs (http://) and mentions (@someone), which are of no use whatsoever.

Emoticons vs. Emojis

Looking further into few records reveals that emojis and emoticons could be useful in predictions. We, humans, do tend to use emoticons and emojis in casual language to express our feelings and state of mind.

On further analysis, we find that tweets do not have any emojis; only emoticons are present. I plan to use an existing library *preprocesssor* to identify and parse emoticons.

What is the difference between Emoticons and Emojis? Citing from the web source – "If you come across a smiley face that contains a character you can find on your computer keyboard, it's an emoticon. If it is a little cartoon (yellowish) figure that is free from the binds of punctuation, numbers, and letters, it's an emoji." Example, :) vs 😊

Data Cleaning

Before we extract useful information from tweets, we need to clean the tweet texts. We conduct data cleaning in two phases – Cursory and Deep Cleaning.

1) In **cursory cleaning**, we remove URLs and mentions right at the start, ensuring we do not delete punctuations until we extract information about (happy and sad) emoticons, which could be useful in prediction later.
2) Extract Emoticons – We extract presence and absence of happy and sad emoticons and save them as *happyEmotsFlag* and *sadEmotsFlag* attributes respectively.
3) In **deep cleaning**, we strive to keep bare essential words in a consistently cohesive form for each tweet. Hence, we not only remove unwanted characters and digits from the tweets, but also apply casing and lemmatization on each token (word).

   We remove the following items from the text.

   a) Emoticons are removed since we have already extracted the required information in two distinct attributes.

   b) Punctuations (! # $? @ % & ( ) * ^_ - + / \ < = > [ ] { } | ~ . " ` ' , : ;) are removed while retaining hashtag text (string following # character), as the latter could be useful for predictions later.

   c) Tabs and line breaks serve no purpose whatsoever, and hence removed.

   d) Numeric digits serve no purpose whatsoever, and hence removed.

   e) Stop words – We remove most common words known as "stop words" using **nltk** library. For the current goal at hand, these stop words would not add any value to the

meaning of the document (each tweet). Generally, the most common words used in a text are “the”, “is”, “in”, “for”, “where”, “when”, “to”, “at” etc.

f) Non-ascii characters are removed. E.g. convert CarolinaåÊAblaze to CarolinaAblaze.

We then apply final transformations.

g) Lower case the characters, so that only one token is created to represent both “Fire” and “fire”.

h) Lemmatize using **spacy** library to converge all the variations of the same word onto one. Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word (lemma). For example, if confronted with tokens *saw, seen, seeing* and *see*, lemmatization would likely return *see*.

Word	**Lemma**
seen/saw/seeing/see	see
drove/drive/driving	drive
better/good	good
playing/played/play	play

Now, we check whether any tweet originally comprised of only extraneous elements, such that after stripping them off, the tweet was left with nothing, and hence, turned to null value. If yes, then we should remove such tweets from the training set.

Fortunately, there are no null strings after data cleaning and transformations. Hence, we move ahead with the entire dataset of 7613 records for statistical modeling.

Statistical Modeling

Since there are myriad ways of tackling an NLP problem, it is usually better to start off with the simplest to the most difficult methodology and then assess which one works best, weighing upon simplicity (or complexity) and performance of the model. As a result, I draw out three different techniques to solve the problem at hand.

i. TF-IDF vectorization + Logistic Regression

ii. Average word embeddings + Logistic Regression/Random Forest

iii. LSTM (Long Short-Term Memory)

While building each statistical model, **log-loss** (aka binary cross-entropy) shall be used as the loss function for each algorithm to optimize while learning from the training data, and accordingly, adjusting the weights or coefficients of the model.

TF-IDF

After available dataset is split into training and test (hold-out) datasets, clean text is used to generate TF-IDF sparse vector for each document. (For more information on TF-IDF, refer to the Glossary). Words (or tokens) that appear in at least 80% of the documents (high-frequency words) or that show up in not more than 5 documents (low-frequency words) are removed before developing a sparse vector of 1783 length for each document. (I played around with these parameters and selected them because they yielded optimum performance later).

Due to huge width of the dataset, I decide to train only Logistic Regression model, which is way faster to run on a high-feature data set than other models such as Random Forest, SVM, etc.

Average word embeddings

Word embedding is a dense vector that represents a word in an n-dimensional space (here, n=300) while retaining its contextual meaning with respect to other words. For example, "king", "queen" and "prince" all appear very close to each other in the n-dimensional space, while showing up farther from words like "nuclear", "rose" and "water".

Using spacy library, word embeddings of each token in a document is listed down. Then, document-level embedding is calculated by averaging embeddings of underlying tokens (words) in the document.

Here is an illustration for averaging 3-dimensional word embeddings.

> Document: "there is fire on the hill"
>
> Remaining tokens after removing stop-words: [fire, hill]
>
> List of word embeddings (spacy): [[3, 6, 2], [9, 4, 6]]
>
> Average embedding of the document: [(3+9)/2, (6+4)/2, (2+6)/2] = [6, 5, 4]

Along with document-level embedding, I also included couple of features to show existence or absence of sad and happy emotions (through emoticons).

With a dense vector of 302 length for each document, I apply two different machine learning algorithms – Logistic Regression and Random Forest, and tune their hyper-parameters.

LSTM

LSTMs are a special kind of RNN (Recurrent Neural Networks), capable of learning long-term dependencies, attempting to model time or sequence dependent behavior. LSTM models consider order of words in a document while statistically learning from the texts.

In a sequence prediction problem such as this one, a sample involves a sequence of input (e.g. time steps) where there are one or more observations at each time step (e.g. features). Therefore, the input training data must be a three-dimensional array with the dimensions ***samples*** x ***timesteps*** x ***features***. Before an LSTM model is trained, we need to ***pre-process*** clean texts to form a 3D array of training data for the model to ingest.

1) Using Keras Tokenizer, build a vocabulary using clean text of training data – pairs of word token and index number. E.g. ('watchout', 9661) and ('zurich', 5628).
2) Using newly-formed vocabulary, convert each document (clean text) into a sequence of numbers, for both training and validation dataset. Basically, replace each word token with the corresponding index value from the vocabulary, yielding a *variable* sequence of numbers with length equal to the number of tokens in the document.

Document (Clean Text)	**Numeric Representation**
detonation fashionable mountaineering electronic watch water resistant couple leisure tab	[406, 2253, 2254, 1577, 50, 110, 1578, 694, 2255, 2256]
respect like talent guess stil human sink low mr affleck	[1243, 3, 2257, 872, 4143, 656, 146, 450, 1244, 2849]
see hurricane guillermo meteoearth	[17, 254, 1836, 2850]
fur leather coat sprite amp weapon choice lifestyle choose back	[4144, 657, 2258, 4145, 6, 77, 1382, 2851, 1579, 40]
heat advisory effect pm pm thursday build heat wave increase humidity lawx	[241, 1132, 306, 70, 70, 630, 531, 241, 115, 1026, 1837, 4146]

   There is an attribute of the Tokenizer object – *num_words* that lets the object choose top *n* most common words to consider while building the numeric sequence of each document. Later, during model training and testing, it would turn out that *n*=6300 yielded the optimum results.
3) Decide the *static/fixed* length of each document that needs to go as input to LSTM-based statistical model. The (fixed) length is set as 26, which is 3 more than the length of the longest document available in the training data.

4) Apply right padding of 0 on each document if the length of the document is less than 26. The idea is to bring each document to a *fixed* length to input training and validation data into LSTM model. (Notice how the variable-length numeric representation above becomes fixed-length sequence below).

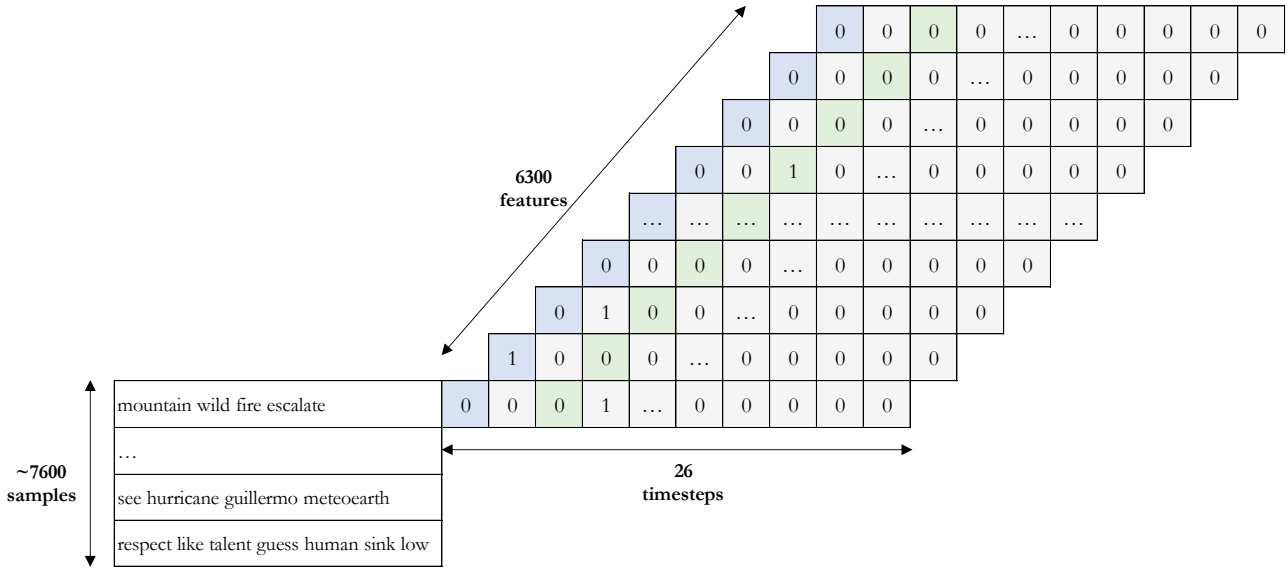
Document (Clean Text)	0	1	2	3	4	5	6	...	19	20	21	22	23	24	25
detonation fashionable mountaineering electronic watch water resistant couple leisure tab	406	2253	2254	1577	50	110	1578	...	0	0	0	0	0	0	0
respect like talent guess stil human sink low mr affleck	1243	3	2257	872	4143	656	146	...	0	0	0	0	0	0	0
see hurricane guillermo meteoearth	17	254	1836	2850	0	0	0	...	0	0	0	0	0	0	0
fur leather coat sprite amp weapon choice lifestyle choose back	4144	657	2258	4145	6	77	1382	...	0	0	0	0	0	0	0
heat advisory effect pm pm thursday build heat wave increase humidity lawx	241	1132	306	70	70	630	531	...	0	0	0	0	0	0	0

5) When the fixed-length sequence (aka training data) is pushed into the learning model, each of the index number is represented as a sparse vector, thereby forming a 3D format essential for the LSTM-based model to take in.

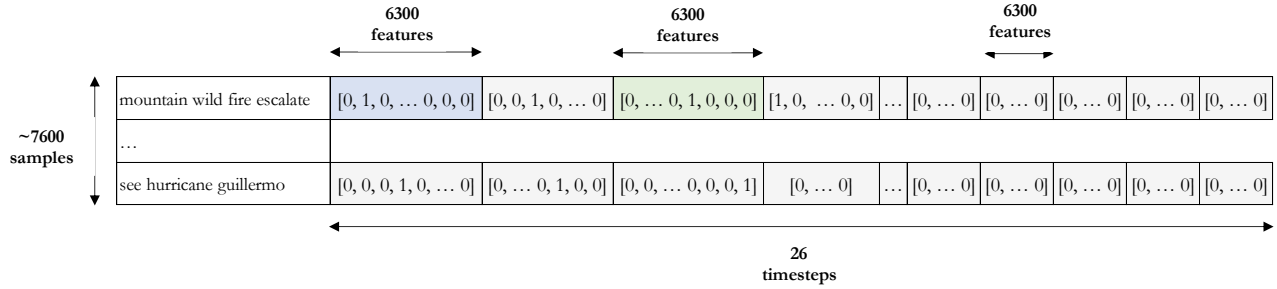
Let's pick up a sample document out of the previous step and see what it would look like in 3D format.

Document (Clean Text)	0	1	2	3	...	20	21	22	23	24	25
mountain wild fire escalate	406	2253	2254	1577	...	0	0	0	0	0	0

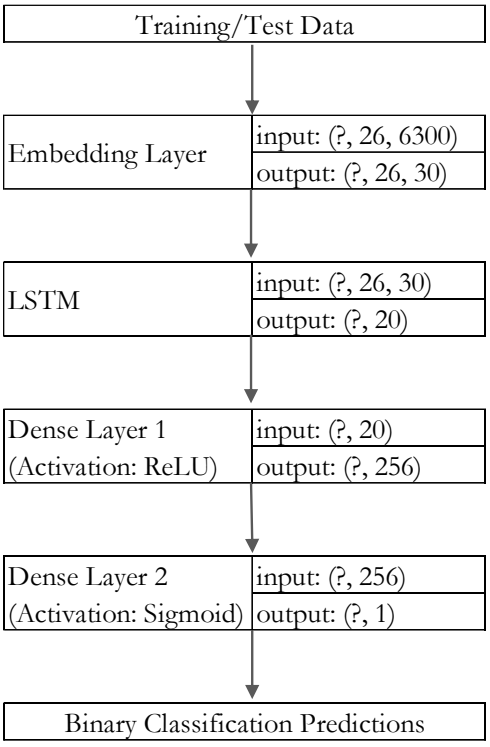
For the current problem, the length of the sparse vector is 6300, which is determined in step 2 by *num_words*. If the token "mountain", indexed as 406, is represented as [0,1,0,0…0,0,0,0] sparse vector, and if "fire", indexed as 2254, is represented as [0,0,0,0…1,0,0,0], then the sample document will look like the following 2D matrix, with blue and green vectors representing "mountain" and "fire" tokens respectively.



When we consider 2D arrays for all samples (documents), we get a 3D format. Hence, **samples, timesteps and features** combined form a 3D matrix for the training data. Alternative way of visualizing the training input data is as follows.



Once numeric representation of the documents is done and the training data ready, statistical model is designed and compiled. The architecture of the implemented model is as shown below.



1) Embedding layer converts sparse vectors into dense word embeddings, reducing length of vector representation of each word token from 6300 to 30.
   The layer takes input data of size (?, 26, 6300) and outputs data of size (?, 26, 30), where "?" denotes the batch size of the learning model. Determined by hit-and-trial, the batch size of 64 turns out to be the optimum one for the problem at hand.

2) LSTM layer takes care of modeling sequence dependent behavior and memorizing order of words in each document. The layer consumes 3D data and outputs in 2D format, releasing 20-long vector for each document.
3) A dense layer, which is a multilayer perceptron (MLP) forming a fully-connected neural network to identify complex patterns in the texts, uses activation function ReLU (rectified Linear Unit) and outputs 256-long vector for each document.
4) The last dense layer uses sigmoid activation function (an s-shaped distribution) to control the output value between 0 and 1, which represents the probability of a document (tweet) denoting a disaster.

Fitting the model requires that we first select the training configuration, such as the number of epochs (loops through the training dataset) and the batch size (number of samples used to estimate model error). For a deeper conceptual understanding of these hyperparameters, please refer to **Epoch & Batch Size** under Glossary section.

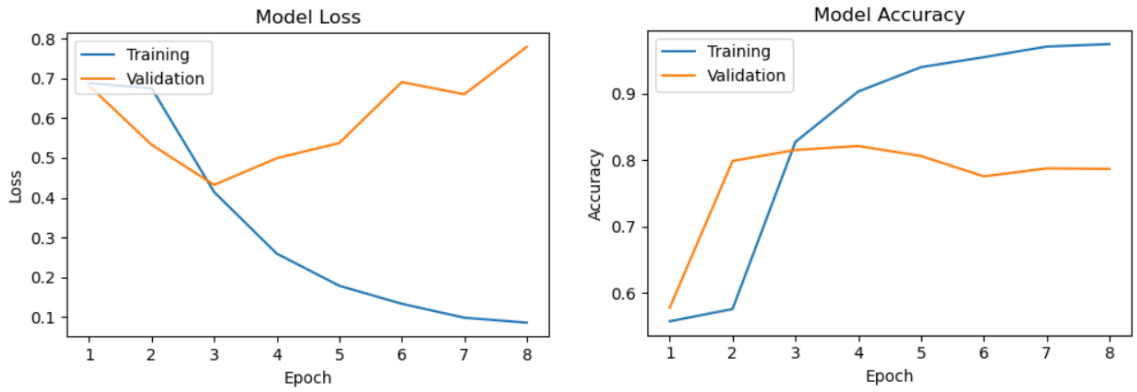
Along with setting up the number of epochs as 20, an *early-stopping criteria* is also established so that the model does not overfit on the training data. Early-stopping criteria allows the learning process to keep monitoring log-loss on validation dataset (and not on training dataset) and stop it if the log-loss does not improve (decrease) substantially for five consecutive epochs.

Following is the consolidated list of hyper-parameters used in the LSTM-based learning process. Each one has been set up to the corresponding value, through hit-and-trial method, to yield optimum results.

a) Fixed-length of each document: 26 (which is 3 more than the longest document's length)
b) Vocabulary size (*num_words*)/length of sparse vector for each word token: 6300
c) Length of word embedding (dense vector) for each token: 30
d) LSTM layer vector: 20
e) MLP (dense) layer vector: 256
f) Batch size: 64
g) Number of epochs: 20 (along with an early-stopping criteria)

Upon training and testing the model, it seems that the model fitting at the end of 3rd epoch yields optimum results. Beyond 3rd epoch, the model treads into an over-fitting territory which is clear from the increasing divide between log-loss scores on training and testing datasets 4th

epoch onwards. Similarly, accuracy of the model on training dataset keeps increasing, while that on testing dataset remains stable 2nd epoch onwards.



Conclusion

Here are the results of each of the vectorization and statistical models discussed above. One thing worth noticing is that all the models fared equally well compared to each other.

Vectorization Model	TF-IDF	Document-level Word Embedding		LSTM
Statistical Model	Logistic Regression	Logistic Regression	Random Forest	
ROC AUC	0.847	0.853	0.858	0.851
Log-loss	0.4685	0.4676	0.4713	0.4579
Accuracy	78.7%	79.1%	81.0%	79.6%
Training Effort (in minutes)	1	1	17	1

While it was possible to get stable results with every run of Random Forest or Logistic Regression model, by using *random_state=101* parameter during initialization of the model, similar stability could not be obtained in the results of LSTM-based model. Though I have listed down one of the best scores out of a dozen of LSTM runs, the results varied in a very narrow range (ROC AUC: 0.845 – 0.854; Accuracy: 78.4%-79.8%).

Considering simplicity of the algorithm and slightly better results among the lot, **Document-level word embedding (aka average word embeddings) with Random Forest** is the clear winner, even though computation effort involved in training and developing the model is considerably high. However, if the size of underlying training dataset were to increase substantially, it would be wise to switch to Logistic Regression model, with either TF-IDF or document-level embedding vectorization, since Logistic Regression is much more efficient, and hence, faster to run on high-feature dataset.

Future Scope

No matter how well the prediction results turn out to be, time limitation on any project opens room for improvement of the predictive results in future iterations. This project is no exception to that belief.

Transfer learning with LSTM-based model could be explored. The existing LSTM-based model deployed an embedding layer to generate dense vector for each token, utilizing contexts of each word as it exists in the tweets. Alternatively, we could leverage existing word2vec embeddings of spacy library (as it has been done in *document-level embedding* algorithm) and plug them into LSTM layer. The updated version of LSTM model could yield better results, potentially surpassing superior results of the incumbent algorithm - *document-level embedding*.

Codebase

All the analysis has been conducted in Jupyter notebook after installing standard Anaconda distribution for Python 3. The entire code has been organized into just one python notebook - *Disaster Tweets.ipynb* - and uploaded on Github repository.

Glossary

Log-Loss

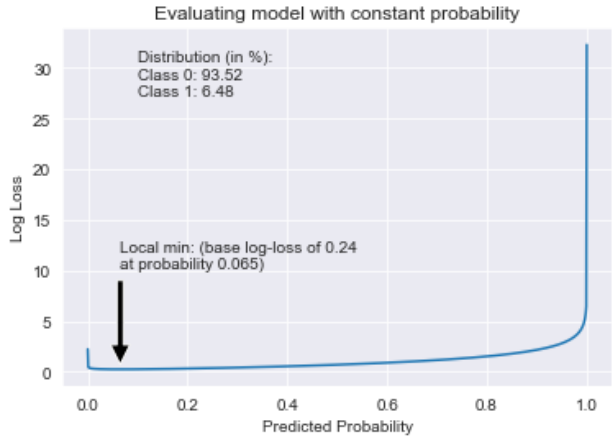
Log-loss score is indicative of how close probability predictions are to the corresponding actual values of 0 or 1. Higher the score, farther the data points from their true values. We can regard log-loss in a classification problem synonymous to MSE (mean squared error) in a regression problem. Both metrics are indicator of how close/far the predictions are from the actual values.

$$Logloss = -\frac{1}{N}\sum_{i=1}^{N}[y_i \, ln \, p_i + (1 - y_i) \, ln(1 - p_i)]$$

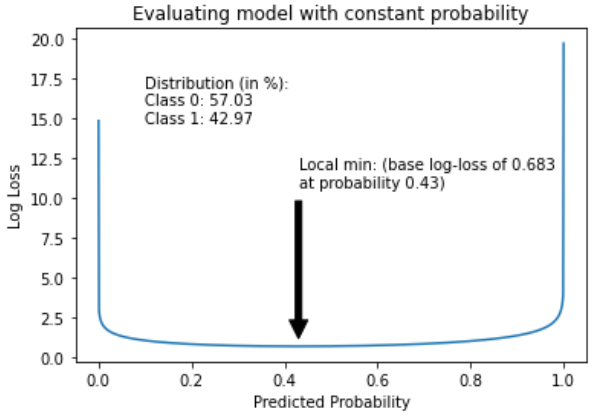
where N is the number of observations, ln is the natural logarithm, y_i is the binary target, and p_i is the predicted probability that y_i equals 1.

A model with lower log-loss value is better than the one with higher log-loss value, provided both the models are applied to the same distribution of dataset. As a result, among a set of models applied on the same validation/test set, the model with *lowest* log-loss score is usually selected as the final model for real-life predictions. Additionally, the chosen statistical model should have a log-loss score lower than the **baseline log-loss** score for the given dataset.

Base log-loss score is determined by the naïve classification model, which simply pegs all the observations with a constant probability equal to % of data with class 1 observations. As shown in the chart below, for the given dataset with 6.5% churn users, log-loss score of 0.24 at constant probability of 0.065 is regarded as the base log-loss score.

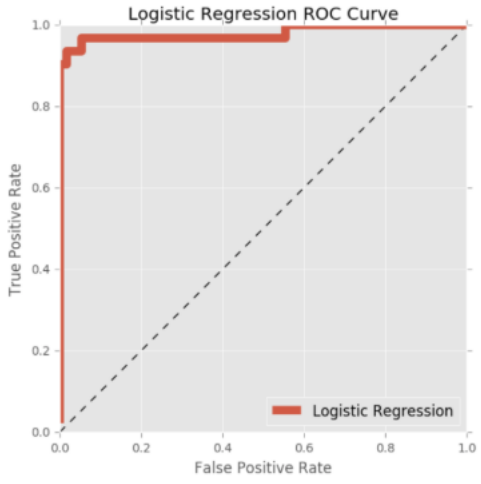


Similarly, for the dataset with 43% of data points under positive class ("1"), log-loss score of 0.683 at constant probability of 0.043 is regarded as the base log-loss score.



ROC-AUC

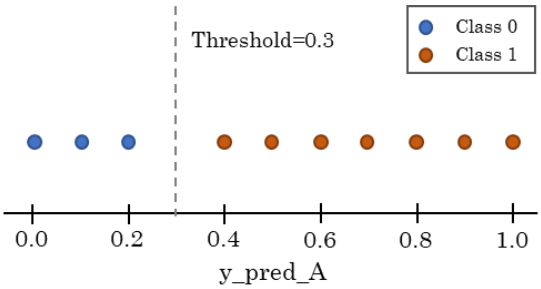
ROC Curve is a commonly used graph that summarizes the performance of a classifier over all possible probability thresholds. It is generated by plotting the True Positive Rate (TPR, aka Recall) on y-axis against the False Positive Rate (FPR) on x-axis as one varies the threshold for assigning observations to a given class.



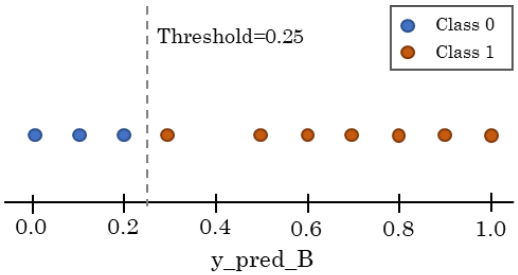
A binary classifier that in fact is just randomly making guesses, would be correct approximately 50% of the time, and the resulting ROC curve would be a diagonal line in which the TPR and FPR are always equal, representing AUC of 0.5. If the AUC is greater than 0.5, the model is better than random guessing. Hence, larger the area under the ROC curve (AUC), better the model is.

ROC-AUC score is indicative of how many data points could be classified correctly upon setting up probability threshold used for classification. Higher the score, higher the number of samples that can be correctly classified.

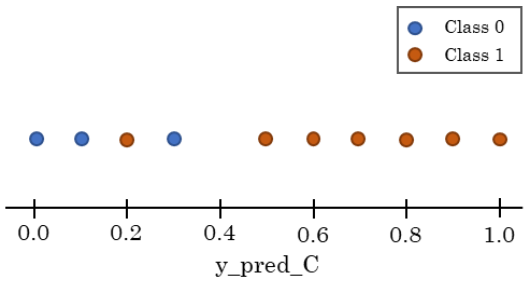
Let's consider 10 data samples, 7 of which belong to positive class "1" and 3 to negative class "0". An algorithm A predicts probabilities for the given 10 samples in the following manner. We can set probability threshold of 0.3 and classify every data point correctly under each class. This would indicate an ROC-AUC of 1.



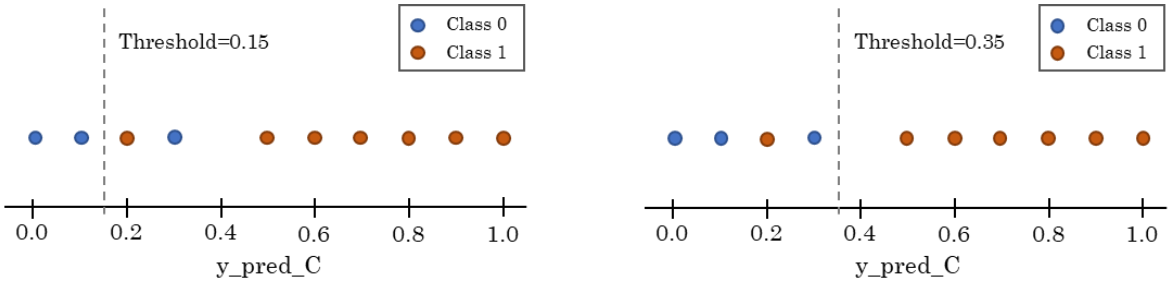
Similarly, given following predictions from algorithm B, we can set the threshold as 0.25 and classify data points correctly. Notice that the fourth data point (from left) garners probability of 0.3 with algorithm B as opposed to 0.4 with algorithm A, forcing us to shift the threshold probability towards left. Despite the shift in threshold probability, we are still able to cleanly classify each sample correctly. Hence, ROC-AUC score is still 1. (Remember that log-loss of algorithm B is more than that of algorithm A, since fourth data point is moving farther away from the true value of 1).



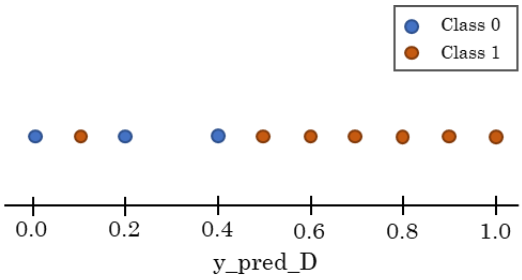
Now, let's say algorithm C predicted probabilities in the following manner, moving fourth data point further to the left by assigning probability of 0.2 and moving third data point to the right by assigning probability of 0.3.



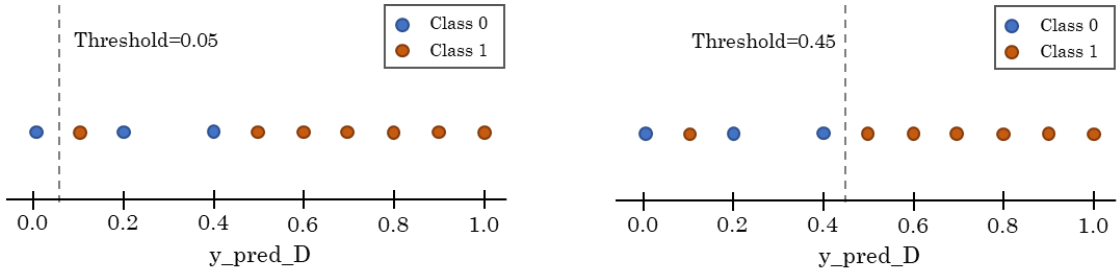
There is no probability threshold that can neatly classify all the data points correctly. If we set the threshold as 0.15 or as 0.35, we get one data point on the incorrect side of the threshold line. Hence, ROC-AUC score is less than 1 (precisely, 0.952).



In the same vein, let's explore following results of algorithm D. Again, there is no demarcation that be done without classifying some data points incorrectly.



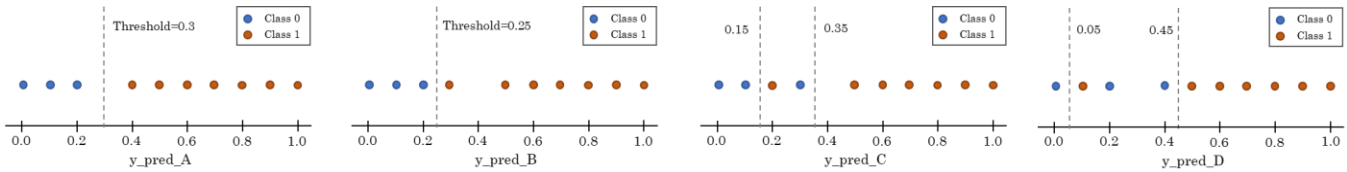
If we choose 0.05 as the threshold, then we risk classifying two data points incorrectly (blue ones on the right of the threshold line). If we choose 0.45 as the threshold, then we classify one data point incorrectly (orange one on the left of the line).



For obvious reason, we would choose threshold of 0.45 rather than 0.05. However, we still can't deny the fact that algo D has the potential to classify a greater number of data points incorrectly. Hence, ROC-AUC of the algo D (0.905) is even lower than that of algo C.

Here is the sample data and predicted probabilities for each data point out of all the algorithms (A, B, C and D), along with scatter plot and ROC-AUC values and log-loss scores of all the algorithms.

y (true)	y_pred_A	y_pred_B	y_pred_C	y_pred_D
1	1	1	1	1
1	0.9	0.9	0.9	0.9
1	0.8	0.8	0.8	0.8
1	0.7	0.7	0.7	0.7
1	0.6	0.6	0.6	0.6
1	0.5	0.5	0.5	0.5
1	0.4	0.3	0.2	0.1
0	0.2	0.2	0.3	0.4
0	0.1	0.1	0.1	0.2
0	0	0	0	0

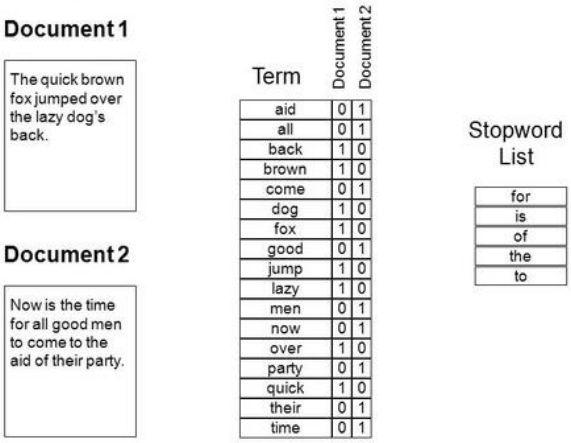


Algorithm	A	B	C	D
ROC-AUC	1.000	1.000	0.952	0.905
Log-loss	0.313	0.342	0.396	0.493

TF-IDF

Vectorization is a general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the

Bag of Words representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.



Each vector can be considered as **bag of words**. Vectors would contain mostly zero values, making them sparse matrices. By itself, these may not be helpful until we consider some statistic, such as presence/absence of a word in a document, count of words (frequency), or TF-IDF (term frequency - inverse document frequency).

There are some potential problems which might arise with the Bag of Words frequency-based model when it is used on large corpora. Since the feature vectors are based on absolute term frequencies, there might be some terms which occur frequently across all documents and these may tend to overshadow other terms in the feature set. The TF-IDF model tries to combat this issue by using a scaling or normalizing factor in its computation.

Instead of counting frequency of occurrence of token, we could also gather TF-IDF statistic. It allows one to determine the most important words in each document, such that most common words don't show up as key words. Each corpus may have shared words beyond just stop-words - ones that should be down-weighted in importance. E.g. “sky” in astronomy corpus. On the other hand, TF-IDF keeps document specific frequent words weighted high.

$$w_{i,j} = tf_{i,j} \times log\left(\frac{N}{df_i}\right)$$

$w_{i,j}$ = tf-idf weight for token i in document j

$tf_{i,j}$ = number of occurrences of token i in document j

df_i = number of documents that contain token i

N = total number of documents

Epoch & Batch Size

In neural networks, epoch and batch size are hyperparameters that are tuned by trying different values and checking what works best for the problem at hand.

A **sample** is a single row of data, including the inputs for the network and the expected output. While an **epoch** means an iteration/loop of the learning model over training dataset, **batch size** denotes number of samples processed, at any time, to estimate model error before the model is updated.

One training epoch means that the learning algorithm has made one complete pass through the training dataset, where samples were separated into randomly selected "batch size" groups (aka batches). The model makes predictions for each sample in the batch, the error is calculated by comparing the prediction to the expected value, an error gradient is estimated, and the weights are updated.

Consider running a deep learning algorithm with batch size of 64 on a training sample of 960 records. Every epoch (iteration/run) shall involve (960/64=) 15 batches. In other words, every epoch shall estimate error, and accordingly, update model weights (960/64=) 15 times before calling it the day. For 100 epochs, the model will be exposed to or pass through the whole dataset 100 times and will process 1500 batches during the entire training process.

While the number of epochs can be set to any integer value between one and infinity, batch size must be an integer between one and the number of samples in the training dataset.

We can run the learning process for as long as we like as determined by the number of epochs. We can even stop the learning process earlier (i.e. before requested number of epochs have run through) by using *stopping criteria* such as a change (or lack of change) in model error over time. For example, we can set the number of epochs for a classification deep learning model as 20 and configure model's stopping criteria such that the learning process will keep monitoring log-loss on validation dataset (and not on training dataset) and stop itself if the log-loss does not decrease by 0.05 for five consecutive epochs. If the process stops at the end of 8 epochs (instead of running for all 20 epochs), we infer that the change in log-loss value is less than the threshold amount of 0.05 from 4^{th} epoch onwards.