Recognizing the Real Disaster Tweets

This project targets using Natural Language Processing methods to create models that can identify if a tweet is referring to an ongoing disaster. The subject was decided in conjunction with the data science resource and competition website kaggle and can be seen at https://www.kaggle.com/c/nlp-getting-started/overview.

Many tweets, often through regular use of hyperbole, will use terms associated with disaster scenarios to talk about everyday occurrences. But if you could mechanically discern the real events from the hyperbole, an automated tweet reader could surface those tweets that refer to actual emergency situations, and help direct rescue responses expediently. With the location information often included in a tweet, such tools could even pave the way for generating hot spot maps to start sending crews out before calls referring to incidents are able to be triaged. Further work could also one day automatically update dispatches already en route to such locations, add tweeted pictures to incident reports, and lower the priority on calls that seem to be originating from an already dispatched to area.

The models to automatically identify disaster tweets would therefore be of import to any emergency response dispatch service. They would be able to deliver aid quicker, and perhaps in some cases even deliver aid where they wouldn't have known to at all, thereby saving costs in property damage, and human life/injury. Since the primary consumers of these models would therefore typically be public services that could not afford to budget for these models, Kaggle is stepping in to recognize the value of competent models for this problem space with a site hosted prize pool.

To train the models for this project, the company figure-eight has provided a dataset to Kaggle of a manually tagged set of ~10,000 tweets in easy to import csv format with each text tweet tagged as disaster or not, and a subset of these tweets also having fields with their contained disaster keywords and location information. In order to engage in the spirit of the Kaggle competition, the goal will be to form an accurate trained model for identifying disaster tweets from just these provided tweets. We therefore won't be wrangling any additional data sets to complete our model, but will perform various preprocessing steps to increase the predictive power of our closed data set.

We'll evaluate various models to determine what has the best predictive capability. We will use a TF-IDF representation of our processed input for standard machine learning approaches, as well as modern NLP techniques with neural networks. We will be evaluating the performance of our models locally via cross validation in the training set, and then additionally via kaggles provided test set which must be submitted for actual F1-score validation as, by kaggle's admission, the test set has had some falsified entries introduced to prevent hardcoding of solutions. Accompanying our kaggle submission will be a github hosted report on our overall process, slides to summarize the same, and jupyter notebooks with further details of our process.

Tweet Pre-processing

For this closed Kaggle NLP problem the primary ask in our data gathering step isn't on amalgamating various sources, but rather on transforming our one data set for maximal utility. While we are provided with some additional information beyond the text on some of the tweets in our data set (namely keywords and location) we'll be disregarding these as the primary intent of our project is to form judgments of the threat level of a tweet based on the text alone.

In approaching our dataset then our first step is to ensure we do indeed have entries in both the text field for every tweet and the label for each tweet: disaster related or not related. Having verified this to be the case we can move on to the steps generally taken in preparing a text corpus for machine learning.

There are a few different general methods with which we can transform text to increase the signal to noise ratio. While some are less useful when the entries in the corpus you're examining are each no longer than a tweet, some steps retain their value. Chief among these is first reducing words to their lemmas, so that different conjugations of a word all have matching representations and can be correctly assigned the same import and predictive value, and second removing so called 'stop words.' These words such as 'the', 'I', or 'as' are liable to appear in any text in great frequency regardless of the general meaning of that text. Since the goal in machine learning for natural language processing is typically to predict the intent of texts, words that would appear along with any intent with no clear relation on how often they would appear should be regarded as not having any predictive power.

Since these are usually the first steps for any NLP problem, there exist a variety of libraries that already reduce words to their lemma and collect the stop words that you may want to remove. We use one of the more widely used python libraries for this process: spacy. Spacy is a comprehensive library that can split up a provided text into 'tokens' and classify each token in various ways. Beyond being able to reduce words to their lemma and match from a pregathered collection of stop words, Spacy in fact stores whole vocabularies and can help us sweep typos from our predictive store, and can also identify punctuation that we'll also wish to remove due to low predictive value.

Another of the reasons for widespread Spacy use is the ability to expand on the default text processing it undergoes. This means we can add another prebuilt library onto the base implementation so that it can also identify 'emoticon' tokens that it would otherwise have thrown out as punctuation, but that quite possibly could have predictive value in our tweet context. Additionally in our unique context we'll encounter words used as hashtags, which could be expected to be used in ways other than the words on their own and we'll want to keep this context. We will therefore be checking the 'left of' token on each of our words as we process them and keeping hashtag words as one word in our final processed tweets.

Before keeping our processed tweets after these removals we also collect and output the most common words among all the tweets to see if we can spot some 'stop words' that Spacy did not.

We end up additionally removing the tokens "u", "2”, "4”, "'s", "s", "|", "amp" and “” from our texts as obviously of low predictive value. With a sanity check of our final tweets we also identify that some hex escape characters are now in our corpus where they weren’t before, and add logic to clean these entries. Finally we spot that Spacy has inexplicably added ‘-PRON-’ to many of our processed tweets and remove this as well.

Topic Modeling

It’s said that we can know a word by the company it keeps. For larger texts this means that we can put together words that often appear together in a text, and pull out certain ‘topics’ about a text as a whole if we can identify a group of words in that text. This sort of modeling can give us useful summary insights (in our case it would be great to be able to pull out ‘disaster’ or ‘safe’), and since it’s generated based wholly on counts of words appearing with other words, we can generate these topic models without our program having to know what exactly it is looking for, and can assign topics to generated groupings after the fact. Training such topics however requires we have enough variety of texts to select over however. And while we make an attempt to create such models on our collection of tweets, our topic modeling appears random at best, indicating that the short length of tweets simply does not provide enough information for this task.

TF-IDF

Having already processed much of the noise words out of our tweets, we can also produce one more useful representation for some of our machine learning models. We know that our machine learning models are agnostic to things like the meanings of words, and will need some kind of numerical representation in order to perform. When it comes to natural language, one commonly utilized vector representation is the bag of words.

To produce these, first the sum of unique words across a corpus must be computed. Then for each document in a corpus your bag of words vector is a vector equal to the length of the total vocabulary (this is partly why removing stop words and punctuation is a helpful first step) with each dimension in the vector corresponding to one of the tokens in the vocabulary, and the number at that dimension for a document being the count of that token in the document (tweet in our case). These vectors are often sparse, and will definitely be so in the case of our short tweets.

Our naive ‘bag of words’ representation can however be improved into a better input model with a little extra computation however. When we begin training our machine learning models we will be trying to determine which dimensions are most likely to influence a label of ‘disaster’ or ‘not disaster’. It’s clear that the words that will be most important for this task are words that strictly appear with one label or the other, and indeed the more often a word appears at all, the less likely it is to be able to strictly define a tweet one way or the other, and the more likely it is that it’s an unidentified stop word.

In order to raise the significance of each of our dimensions, we will transform them with a method known as Term- Frequency / Inverse Document Frequency (TF-IDF). With this technique each term is weighted not just with its appearance count (Term Frequency) in a document, but also how many documents it appears in as a whole. The 'Inverse Document Frequency' comes into play with a term being degraded in weight the more documents it appears in. This gives us proportional values for each term that better represent their predictive values, and should produce more reliable models with standard machine learning techniques.

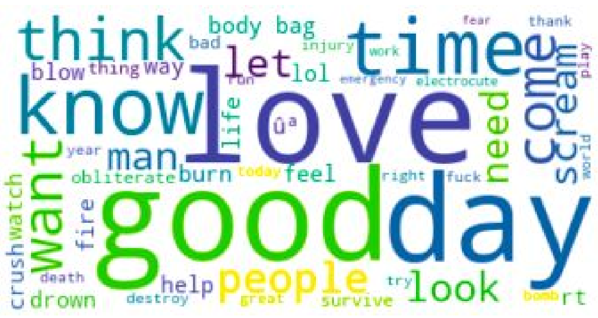
Tweet Exploration

In the world of NLP there are limited exploration steps that we can take to describe our problem space. Indeed everything comes down to how particular words (or sometimes statements) map to particular intents.

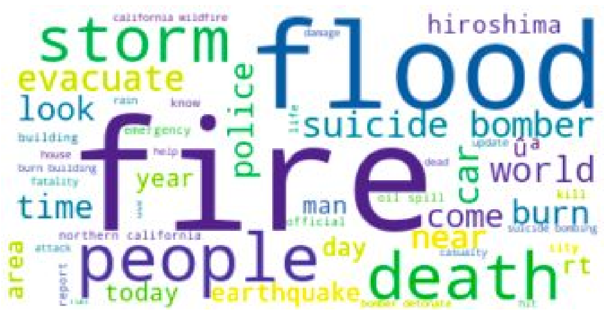
Word Clouds

In our case, short tweets mean words are the size of our interest, so we wish to visualize what words align to tweets that are 'neutral' vs those that refer to an ongoing disaster, as identified by the company figure 8 with our data set. Using our already preprocessed set of tweets, we can combine the text of all 'neutral' tweets together, and all 'disaster' tweets together to form 'word cloud' representations, in which we print all of the most commonly appearing words in a graphic visual, but have greater font sizes for words that appear relatively more frequently in the collection. For each data set this yields:

Neutral



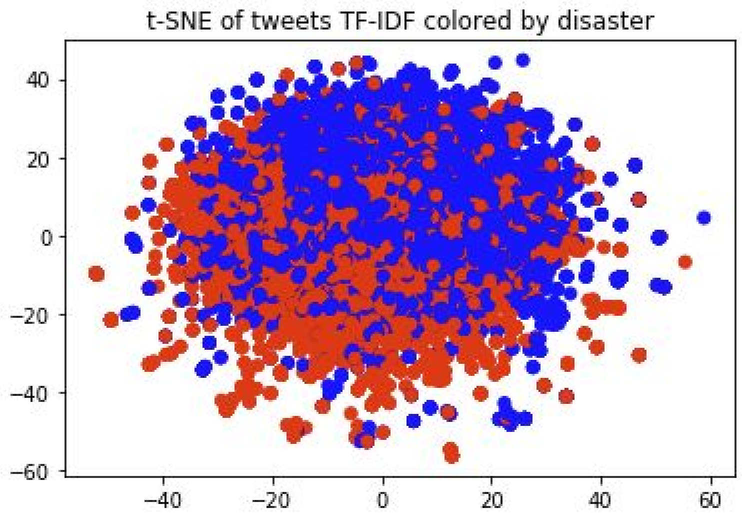
Disaster



Initial takeaways are that it's odd to see 'body bag' in the neutral collection of tweets, and that there's probably a little more cleanup we want to go back to before passing into our machine learning models. Namely, we first want to remove the non interpretable characters (ûª), and judging influence of tokens by their relative sizes, we want to remove 'rt' across the board which we see being nearly equally represented in both data sets. We revise our preprocessing notebook accordingly and going forward with machine learning will use the tweets with these removed.

SNE

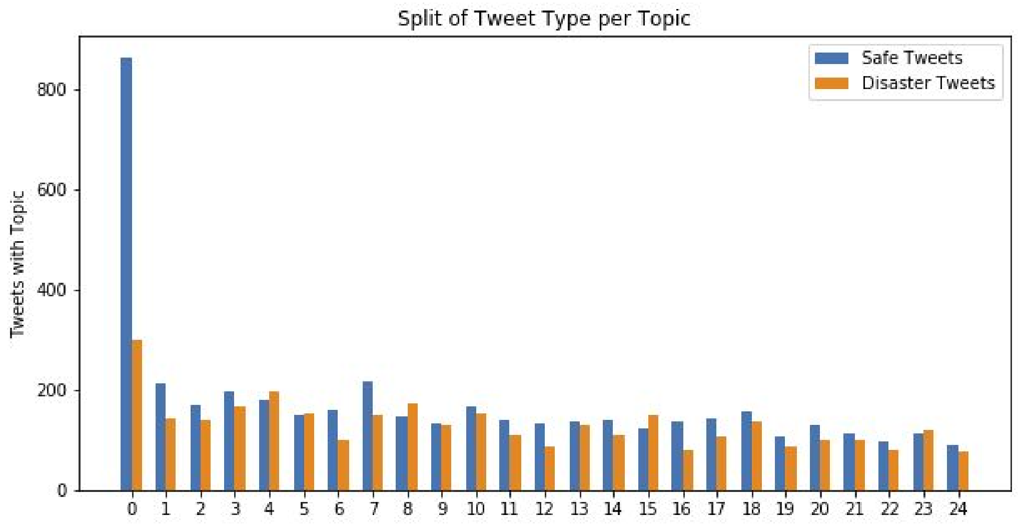
Another method to visualize if there are distincitive differences between two different corpuses (in our case, disaster tweets and non disaster tweets) involves the usage of our TF-IDF representation of our tweets. Via an algorithm known as SNE we can visualize the high dimensional spaces (our TF-IDF vectors) in just two dimensions, with the algorithm focusing on preserving the effective distance between all points in a collection as it's goal when transforming down to two dimensions. If we color the points in our two different data sets differently then, we may be able to see if there's a difference at a glance between the two corpuses. Applying the algorithm in this way arrives at, and coloring 'safe' tweets blues and 'disaster' tweets red arrives at



And while it's not a distinct separation, we can spot a some separation about the diagonal from top left to lower right, which gives us some hope for the separability of our two classes in machine learning.

LDA

Finally we turn to our LDA topic modeling. We recall that we didn't have much faith in the usability of this algorithm due to the short length of the documents in our corpus. We can still view counts of each tweet in each of our modeled topics however, and doing so arrives at:



which confirms that no topic is distinctly ‘safe’ or ‘disaster.’ Topic 0 might earn our curiosity though due to the difference in representation, but looking at the words gathered in this topic shows us it’s not exactly a ‘safe’ set:

- like
- survive
- suicide
- wreck
- day
- kill
- think
- war
- traumatised
- good
- police
- let
- come
- bombing
- bad
- car
- stop
- crash
- kid
- zone
- look
- need
- 70
- watch
- survivor

So we conclude these topics are not useful for the machine analysis we will be turning to next.

Machine Learning

We would now like to develop some models that can be used to predict whether a tweet is related to an ongoing disaster or not without human input. We will aim to train a few different models on a training subset of our input data, and judge the performance of our trained models on hold out testing data sets. We will train four different classification models to predict either ‘disaster’ or ‘safe’: a random forest, a gradient boosted ensemble, a naive bayes classifier, and a neural network with an LSTM layer. Our recommendation would then be to move forward with improvements on the model that scores the highest accuracy of the ones we train.

Random Forest

Due to the high dimensionality of our TF-IDF input, standard logistic regression is unlikely to perform well as a classifier and while certain kernel functions may allow a functional model to be produced with SVM, a more appropriate choice would be to fit non-linear classification model.

The first such model we choose to fit is the Random Forest ensemble model which is one of the most popular, and proficient, prediction models used in problems where a linear solution is intractable such as in this case.

The model utilizes the standard decision tree creation mechanism where a training set of data is split by picking thresholds on one dimension of the data at a time that best divide given training data into separate classes. And then those split off groups are split again until final decision paths are discovered that can be applied to unseen data. Where random forest deviates is that it makes many trees instead of just one, each one being distinguished by using bagging to create a different set of data from the same set of training data. Then predictions are made from a majority vote of the final set of trained trees.

There are several tuning choices that can be made to potentially improve on the final ensemble. Due to the limits of our processing time, we only tuned two for this project. First, we tried several different ensemble sizes, and then perhaps more crucially, we also aimed to tune the amount of variables each tree had access to when training through random selections of subsets of the dimensions for each tree. This serves to cause trees to train on information that could have predictive value, but would be passed over when other variables were available. After a grid search on these two parameters we arrive at an accuracy of 0.79 for our best random forest model which uses 100 estimators and log2 of the full features available in each estimator.

Gradient Boosted Trees

Gradient boosting is another form of decision tree ensemble gaining in popularity as the final models frequently outperforms the random forest. Unlike random forests each tree in a gradient boosted ensemble operates on the full set of data. The difference then between trees is derived by each tree being sequentially trained on the residuals of the predictions of the ensemble up to that point, rather than each trained in seclusion.

The particular implementation we choose to use is the XGBoost library, quickly becoming a front runner in the machine learning world as a whole, which adds a regularization parameter, sometimes prioritizing models with slightly less good fit if achieved through fewer parameters. Gradient boosted trees also can be tuned in a variety of ways, and so again we choose a subset of them. The learning rate which determines how much each additional tree weighs in the final ensemble we leave fixed at the commonly well performing 0.1 with 1000 estimators, and instead choose to grid search over the max_depth of each tree, the min weight a child node must have in the final ensemble to be split, and the subsample of the training data to use in training each tree. Our best tuned model ends up using a max depth of 5 for each tree, a min child weight of 1, and the full sample of data, to predict with an accuracy of 0.78

Naive Bayes

We also train a non ensemble based additional linear classifier to determine its performance. Naive bayes is another method to produce a non linear model and can therefore still perform on larger dimensional spaces. The method follows from bayes theorem of probability that relates the probability of a certain outcome given a certain prerequisite by the relative occurrence of that prerequisite and the occurrence of the two events together. Estimates from this method then are heavily reliant on how well the training set of data matches the real space you are trying to model. The naive in the method name refers to the fact that when training the model every dimension is considered individually whereas in actuality certain dimensions are likely to be reliant on each other. We use sklearn's gaussian naive bayes implementation which allows non whole number inputs on our TF-IDF input and have no hyperparameter tuning to perform in this case. Attempting to learn a model for prediction with naive bayes ends up producing an accuracy of 0.67, suggesting it is inferior in our use case to the trees we have already trained.

Neural Network

Finally we take a neural network approach to our problem to see if we can outperform the more generalized models. Neural networks are usually a little less clear in mechanism, and are somewhat more complex in setup as well. This is partly because they refer more to a whole space of model design than any singular approach in themselves, but they generally can be visualized somewhat like the human neurons they're named after in which learning is accomplished by simple systems of positive and negative reinforcements. A neural network can be seen as a dense graph then, best pictured in a series of layers. The first layer is an input of arbitrary length over arbitrary dimensionality with a node for each dimension's value. Following layers can be of arbitrary size, but usually have every node connected to every node in the previous layer by a separate weight on every connection, and then summing at every node in the layer and possibly performing some limiting function, before each node densely connects to the next layer with a different weight on each connection. We finally terminate in a layer that just determines our classification. With this setup we can actually assign the required weights on our connections somewhat at random to start, and use back propagation of gradients on some error function on final predictions to adjust the values of each connection's weight leading up to the final node until relatively good prediction models are achieved. This can require a good amount of iterations until the model performs well however, and therefore a decent amount of compute power can be required up front to train models, but following this the models are usually fairly quick predictors.

In the world of NLP specifically, one of the common uses of these neural networks is to "learn" a sort of numerical relation between the meanings of different words. The idea is that if words frequently occur in similar contexts, they probably have similar meanings. Since a neural network simply learns over iterations to predict an output from an input somewhat arbitrarily, if you set an output of two classes, that two words occur in the same context or they don't, and begin training iterations with input of two words in the vocabulary space of our tweets, the weights for similar words to the same layer nodes start to resemble each other. This can then not only help you discover word similarities, but provide a better than random starting point for other language classifiers.

In our neural network exercise, we will be using the keras library for our implementations, and we choose to first iterate over our tweets to produce the 'word vectors' layer for words that appear close to each other that was previously mentioned. We then create another model for the prediction that we are interested in, 'disaster' or 'not' starting with these weights. We use one more mechanism however that distinguishes neural networks from the TF-IDF input representation we had used previously. With neural networks we can actually feed a sequence as an input, so our model can actually start to learn context and the relations of words being in a certain order affecting the final classification of a tweet. With back propagation however this can result in a diminishing or exploding gradient problem with each new word in the sequence before the final classification. A certain type of layer has been defined that helps regulate this problem known as LSTM and we again use keras' implementation to introduce such a layer into our neural network before the final classification. After all training is said and done, our final produced neural network is able to classify tweets in our test set with an accuracy of 0.78.

Our final accuracies for each of our models then are:

Random Forest	XGBoost	Naive Bayes	Neural Networks
0.79	0.78	0.67	0.78