Classifying Mutations by Text

Maxwell King and Seth Pate

CS 5100

14 December 2018

Problem Statement

We are participating in a (now closed) Kaggle competition: Memorial Sloan-Kettering: Redefining Cancer Treatment. We are challenged to use machine

learning to classify a genetic mutation into one of nine categories, given medical literature written about the mutation. MSK sponsored this challenge in order to reduce costs in personalized drug development (explaining the first-place prize of $15,000).

To treat a disease like cancer with personalized medicine, doctors take a biopsy of the tumor and sequence its DNA to identify mutations that might be driving the tumor. A mutation is a variant form of a gene.[1] As a single tumor can contain thousands of mutations, doctors must discriminate between harmful and less dangerous mutations. Oncologists have developed a system of nine classifications to sort mutations in this way, ranking them from most (I) to least (IX) dangerous.[2]

Currently, researchers must consult medical literature to classify mutations. If a machine learning algorithm provided with medical text could recreate the performance of these researchers, doctors could identify target mutations more quickly and cheaply, speeding the development of personalized medicine.

Technical Problem Statement

This is a text classification problem, where we are given text samples, and asked to predict which class a sample should be assigned.

Input & Output

Our **inputs** will be raw text for each sample. We also have classification data (a value from 1-9) for each sample in the training data. We will extract **features** from this text data, and use the training classifications to fit **estimators** to those features as part of a regression.

Using this estimators, our model can **predict** classifications for novel text data (the "test" set)[3]. For each sample, our **output** will be nine numbers: the probability that the sample should fall into each of the nine categories. These probabilities represent the model's confidence in its judgment.

Evaluation

A simple way to evaluate the performance of the model would be to calculate its **accuracy**, the ratio of correct classifications to total classifications. However, accuracy

[1] Certain variants are more common than others, but "normal" is a difficult concept in biology. Mutations are in the eye of the beholder.
[2] This is a simplification, but classifying tumors properly is out of scope for this paper.
[3] Unless otherwise specified, "test" data refers to a portion of training data that we have set aside for that purpose. See "Data" section below.

would not distinguish between a model that was both correct and extremely certain of its correctness, and a model that was correct but much less sure.

To account for this, we will also calculate the **log loss** between our model's predicted classifications and the actual classifications over all samples.[4]

Mathematically, log loss is

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij}$$

(in which **N** is the number of samples, **M** is the number of classifications, **y** is one if the classification of sample **i** to class **j** is correct and zero otherwise, and **p** is the model's confidence in assignment **i** to **j**)[5]

The ideal model would produce a log loss of zero, since it never made an incorrect call. However, mortal models will be penalized for incorrect calls. Specifically, if a model gave a very low probability (.1) for the correct classification, the log loss would increase dramatically (-log(.1), or 1). A perfectly bad model would have infinite log loss.

Rather than rely on Kaggle's evaluation function, we will evaluate our own model by training it with **cross validation**.[6] Validation is a technique of reserving some training data to "validate" the model, ensuring that we did not overfit on the test set when selecting hyperparameters.[7] Cross validation allows us to perform validation without reserving valuable training data as a validation set.

In **3-fold** cross validation, the training data is divided into three "folds", and training takes place in three rounds. In each round, one fold serves as the test data, and the model is trained on the other two folds. This approach ensures that each sample is only in the "test" set once, and therefore that the resulting estimators and hyperparameters are not dependent on which data was randomly chosen to train the model.

Because our classes are not evenly distributed (see the "Data" section below), we will use **stratified** cross-validation. In this approach, each "fold" has an even proportion of classes in it (see "fig. A" below).

---

[4] https://scikit-learn.org/stable/modules/model_evaluation.html#log-loss
[5] https://www.r-bloggers.com/making-sense-of-logarithmic-loss/
[6] https://scikit-learn.org/stable/modules/cross_validation.html#stratified-k-fold
[7] In order to use cross-validation, we are assuming that our samples are independent and identically distributed (i.i.d.). That being said, given that medical research is ongoing, it may be wiser to use a "time series-aware cross-validation scheme" (see "Analysis" section below)

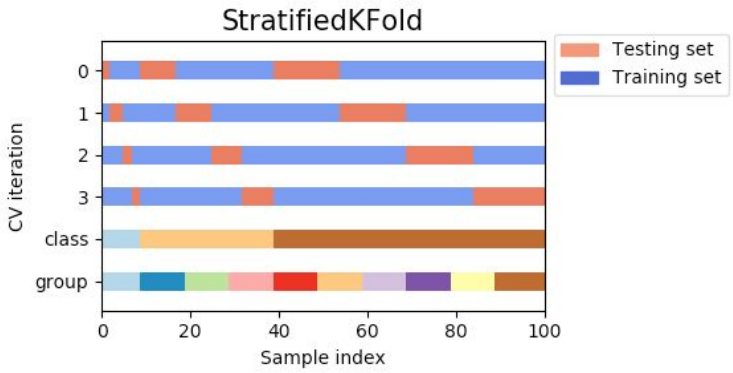

fig. A: Stratified K-Fold (source: scikit-learn.org)

After performing cross validation, the model will provide the nine predicted probabilities for each sample, according to the round in which the sample was in the test set. We will use these probabilities to calculate log loss and thereby evaluate our model.

Dataset

Memorial Sloan-Kettering Hospital provided the dataset on Kaggle. The dataset is split into "training" and "test" groups. There are 3320 samples in the training data, and 5667 in the test data.[8] Because we decided not to use Kaggle's submission feature, we are not using the provided test data. Our model is evaluated solely on reserved training data.

The data is divided into two parts: a "variants" file and a "text" file. The *variants* file contains reference information about each sample:

```
ID,Gene,Variation,Class
0,FAM58A,Truncating Mutations,1
1,CBL,W802*,2
2,CBL,Q249E,2
3,CBL,N454D,3
4,CBL,L399V,4
```

fig. B: "training_variants.txt"

As shown above, samples are organized by a positive integer, "ID". The gene-variant pair is given, and in this training data file, the oncologist's classification of the resulting mutation is provided.

The *text* file is simple a pipe-delimited file of medical literature excerpts relating to each mutation, sorted by "ID":

[8] The imbalance is because some of the test samples were machine-generated, in order to prevent hand-labeled solutions.

```
ID,Text
0||Cyclin-dependent kinases (CDKs)
regulate a variety of fundamental
cellular processes. CDK10 stands out as
one of the last orphan CDKs for which no
activating cyclin has been identified and
no kinase activity revealed. Previous
work has shown that CDK10 silencing
increases ETS2 (v-ets erythroblastosis
```

fig. C: "training_text.txt"

Note that, as you can see in fig. C, medical literature has an extremely diverse dictionary, frequently including words like "*erythroblastosis*" and abbreviations like "*ETS2*". Intuitively, this could complicate regression (it is harder to predict which words are meaningful) or aid it (greater word diversity may allow us to select certain words that are strongly associated with only one or two classifications).

Analysis of Data

We are very grateful to certain Kaggle contributors who made reported their initial observations about the data.[9]

Gene		Variation		Class	
BRCA1	: 264	Truncating Mutations	: 93	1	:568
TP53	: 163	Deletion	: 74	2	:452
EGFR	: 141	Amplification	: 71	3	: 89
PTEN	: 126	Fusions	: 34	4	:686
BRCA2	: 125	Overexpression	: 6	5	:242
KIT	: 99	G12V	: 4	6	:275
BRAF	: 93	E17K	: 3	7	:953
ALK	: 69	Q61H	: 3	8	: 19
(Other)	:2241	(Other)	:3033	9	: 37

fig. D: frequency of genes, variants (source: kaggle.org)

Here, we see that BRCA, a well-known gene strongly associated with breast cancer, makes up a disproportionate number of mutations. In the right-hand column on variants, we see that generic mutations (such as the deletion or overexpression of a

[9] This R user (https://www.kaggle.com/headsortails/personalised-medicine-eda-with-tidy-r) was particularly helpful, and we reprint several of their graphics here.

gene) are much more common than more specific mutations (E17K, which may represent a more subtly modified gene).

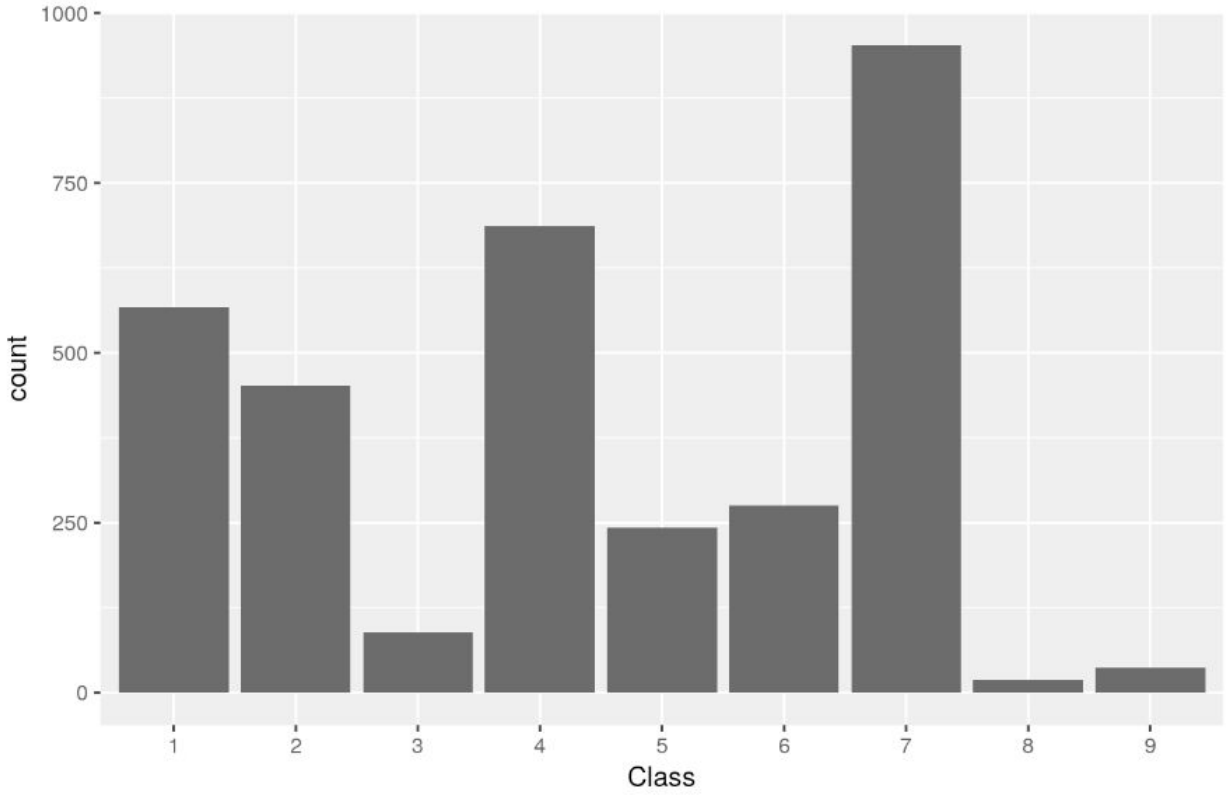

fig. E: histogram of classifications (source: kaggle.org)

Many mutations fall into Class VII, a relatively benign category, supporting our intuitive belief that most mutations are not causing dangerous tumors. However, classes I and II are also well-represented, implying that malignant mutations are well represented in medical literature. Our model had a hard time predicting classes VIII and IX, given that there are relatively few samples of those classes in the training data.

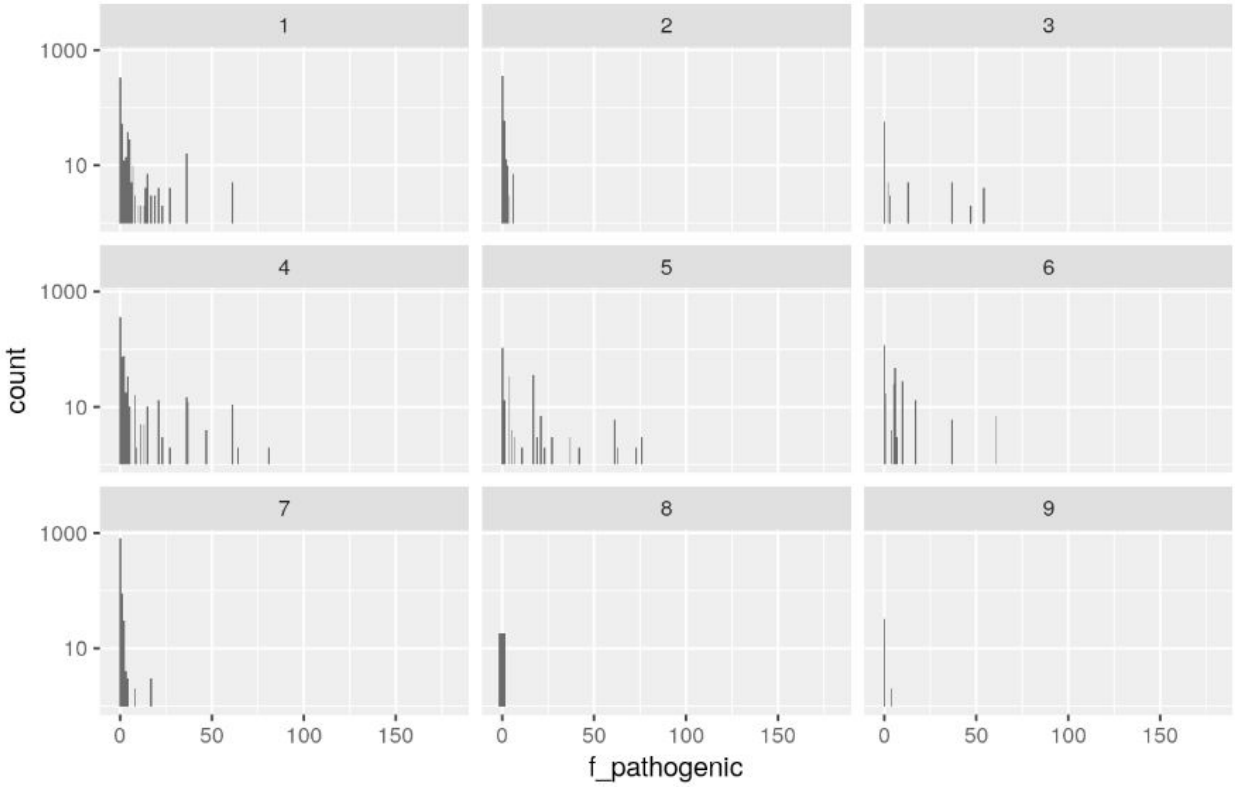

fig. F: histogram of the word "pathogenic" in samples, by classification (source: kaggle.org)

Users found some insights by making basic observations about the frequency of words according to the classification of the mutation. For example, "pathogenic" appeared most frequently in Class I mutations--the most dangerous category. However, the author of fig F. notes that a simple word frequency approach does not differentiate "pathogenic" from "not pathogenic". We will discuss techniques to address this issue in "Feature Generation" below.

Finally, users noted that there are duplicate entries in the training_text.txt file. This is because each mutation consists of a *variant* on a *gene*. Some journal articles are published about a gene and contain information on several different variants. Our feature generation methods didn't allow us to take this information into account, but users reported good results using Long Short Term Memory (LSTM) neural networks and coding features according to their *gene* and *variant* label.[10]

Even after exploring the data, we lacked the domain knowledge to do much pre-processing on the data. We were limited to the simple fix of removing samples with *null* text entries. This done, we moved on to generating features from our text files.

[10] https://www.kaggle.com/reiinakano/basic-nlp-bag-of-words-tf-idf-word2vec-lstm

Feature Generation

We generated features from the text of each sample to give useful information to different algorithms so that they could fit estimators and predict the classification of new samples. This process turned out to be limiting factor in the project. The results of the predictions depended heavily on the features that were extracted from the text.

Bag of Words

We began with a naive approach called "Bag of Words", which generates features according to unique words and how frequently they appear in a sample. This is a common (and simple) approach in text classifications, so we chose to begin with it and iterate.

First, each string set is broken down into its individual words.

```
'Final AI Project' -> ['Final','AI','Project']
```

A project level dictionary is made containing every word in the text. A new array of features is generated, containing, for each sample, an array of how frequently each word in the dictionary appeared in that sample.

```
Bag := [['Final','AI','Project'],['Bag' 'of' 'AI']]
Dict := ['Final','AI','Project','Bag' 'of']
Vecotrized := [[1,1,1,0,0],[0,1,0,1,1]]
```

In the example above, feature counts of "0" are stored. However, since the dictionary was quite large in this case, we appreciated that scikit-learn tools store feature counts in a sparse array, omitting 0s.

This basic concept of breaking the sample into subsections and using those subsections to generate features about the larger sample was the basis for all our feature generation strategies in this project.

Stop Words

We know that not all words are useful to count. Words such as "*the*" and "*a*" don't provide any meaningful information to a classifier, so we wanted to avoid incorporating them into our features. Scikit-learn supports lists of "stop words" which it will ignore when generating features. We used their default list of English stop words.

However, we learned that stop words *do* provide meaning when breaking up text into n-grams rather than counting single words. As referenced above, "not pathogenic"

has a very different meaning than "pathogenic". When using an n-gram approach to features, we didn't use a list of stop words.[11]

Term Frequency – Inverse Document Frequency (TFIDF)

We learned that a simple word count suffers from two problems. First, not all text samples are the same length, so word counts should be normalized somehow across all samples. Second, rare words like "*erythroblastosis*" may be more significant to classification than common ones like "*diagnosis*", but simple counts don't include this information.

A common solution is to transform word counts with "Term Frequency - Inverse Document Frequency".[12][13] The TF-IDF value for a document *d* and term *t* is the term's frequency multiplied by the inverse of its overall document frequency, written as:

$$tfidf(d,t) = tf(t) * idf(d,t)$$

Term frequency is simply calculated by dividing the frequency by the number of words in the document. Inverse document frequency is given by dividing the number of documents by the number of documents in which the term appears at least once:

$$idf(d,t) = \log \frac{1+n}{1+df(d,t)} + 1$$

Note that the formula above incorporates smoothing by adding 1 to the numerator in the denominator, preventing zero division.[14]

Sublinear Text Frequency Scaling

To further reduce the weight of extremely frequent terms, scikit-learn offers a "sublinear tf scaling" method. Instead of calculating term frequency (tf) as above, sublinear scaling uses:

$$tf = 1 + \log(tf)$$

We used sublinear scaling whenever we used TF-IDF to transform feature counts.

---

[11] We actually did use stop words to create bigram features and applied those features to a logistic regression. The regression tended to guess that every sample was class VII (the most frequent class)--a good guess if you don't know anything else.

[12] https://nlp.stanford.edu/IR-book/html/htmledition/inverse-document-frequency-1.html

[13] https://scikit-learn.org/stable/modules/feature_extraction.html#tfidf-term-weighting

[14] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html#sklearn.feature_extraction.text.TfidfTransformer

Singular Value Decomposition

Singular Value Decomposition is a sophisticated technique meant to address two (related) problems: synonymy and polysemy, both of which roughly mean that different words can mean the same thing. SVD (also called Truncated SVD) attempts to reduce the dimensionality of feature vectors to account for these two problems.[15][16] When used in a text context, SVD is referred to as "Latent Semantic Analysis".[17] SVD had mixed results on the accuracy of the learning algorithms in this project.

We applied SVD because more experienced users on Kaggle recommended this technique.[18] However, it relies on matrix math that we don't really understand.

Bigram Tokenization

So far we had only used the 'Bag of Words' tokenization method in which the tokens were single words. These implementations did not yield great results, so we tried another tokenization method, bigrams, a specific form of n-gram tokenization. This tokenization counts not only words, but rather single words and word pairs. We hoped that bigrams would help us solve our earlier problem of differentiating "pathogenic" from "non-pathogenic".

```
Input :=  ['Final AI Project Bigram']
Dict := ['Final','AI','Project','Bigram','Final AI', 'AI Project', 'Project Bigram']
```

This method of splitting up the text gives more features and incorporates information about neighboring terms. After it generates counts, those counts can be further transformed with many of the same methods described above (TFIDF, SVD). However, as mentioned earlier, using "stop words" can remove the benefit that bigrams might provide.

Because bigram feature generation took a long time, we reserved it for our best-performing algorithm, discussed below.

Algorithms

After generating features, we used several algorithms to fit estimators and predict classifications of test data. Initially, we experimented with a Multinomial Naive Bayes

[15] https://scikit-learn.org/stable/modules/decomposition.html#lsa
[16] https://nlp.stanford.edu/IR-book/pdf/18lsi.pdf
[17] https://scikit-learn.org/stable/modules/decomposition.html#truncated-singular-value-decomposition-and-latent-semantic-analysis
[18] https://www.kaggle.com/reiinakano/basic-nlp-bag-of-words-tf-idf-word2vec-lstm

algorithm. However, it performed poorly, and research (as well as the example of other users on Kaggle) led us to try different types of classifiers, including Logistic Regression, Stochastic Gradient Descent, and Random Forest.

Multinomial Naive Bayes

The Naive Bayes classifier makes the simple assumption that all features are independent given the class of the sample. Scikit-learn includes a multinomial classifier that was suitable for text analysis.

Results

Our implementation of Naive Bayes did not easily allow us to calculate log-loss. However, our accuracy was low (about 25%), and comparison to other Naive Bayes models from other Kaggle users show that their log loss was over 15 (quite high!).[19] This led us away from Naive Bayes as a solution.

Logistic Regression

The next algorithm we tried to classify samples was the standard logistic regression algorithm in scikit-learn. A logistic regression made sense, given that we were interested in returning probabilities per classification for each sample. This algorithm implements the same logistic regression algorithm we learned in class, using the sigmoid function (or what scikit-learn calls the logistic function).

Results (with word frequencies)

From running logistic regression with the standard bag of words tokenization method the following results were found.

Algorithm: Logistic Regression | Tokenization: Bag of words				
	Raw Count	Truncated Raw Count	TFIDF	Truncated TFIDF
Log loss	8.9788	6.2173	1.6269	1.6157
Accuracy	0.3348	0.1915	0.3978	0.4095

When combined with features generated by Truncated TF-IDF, logistic regression gave us some of our best results. The matrix below shows that it did a decent job at accurately representing the data from the training set. From the initial reading of the raw data, we saw that classes I, IV, and VII were the most common classifications, and this algorithm was reasonably good at identifying those classes correctly.

[19] https://www.kaggle.com/merckel/nci-thesaurus-naive-bayes-vs-rf-gbm-glm-dl

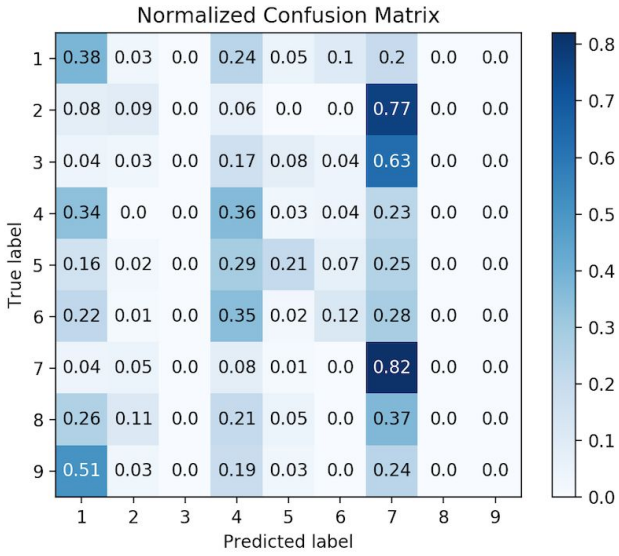

fig. H: Confusion Matrix for logistic regression with TF-IDF features

Results (with bigrams)

Bigram extraction was most successful when combined with Truncated TFIDF.

Algorithm: Logistic Regression | Tokenization: Bigrams				
	Raw Count	Truncated Raw Count	TFIDF	Truncated TFIDF
Log loss	8.2068	5.7587	1.6909	1.6774
Accuracy	0.3502	0.1930	0.3538	0.3821

As seen below, bigram features led the classifier to simply place everything in the most common class. We expected our regression to perform better with bigrams, not worse! Perhaps bigrams created too many features for the regression to distinguish meaningfully between them. It would have been interesting to compare bigram features with other algorithms, but we didn't have time to support this.

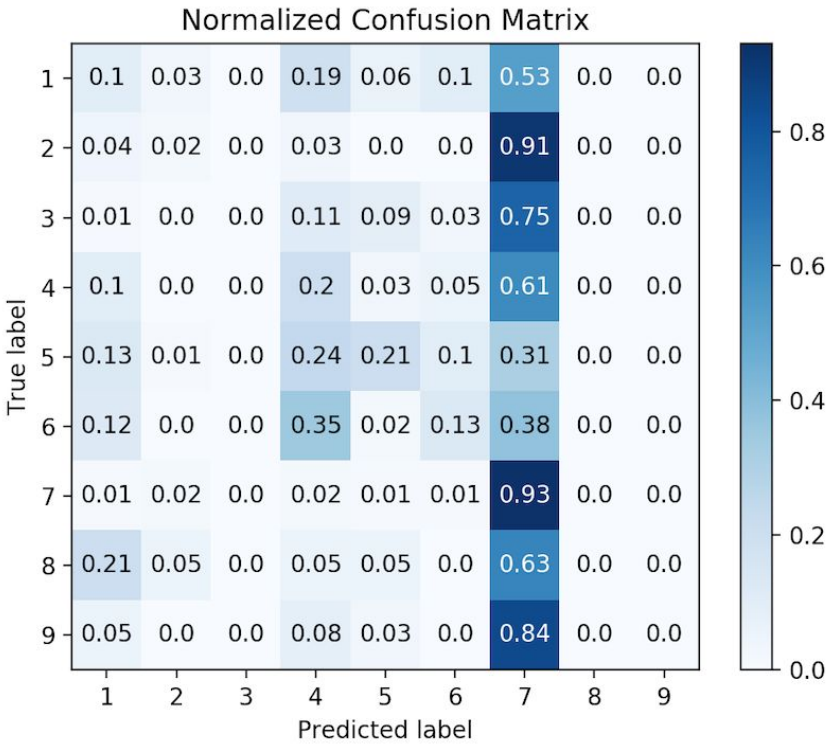

fig I. Confusion Matrix for logistic regression with Truncated TF-IDF bigram features

Stochastic Gradient Descent with Logistic Regression

We also used a stochastic gradient descent algorithm with a log loss function, giving us logistic regression. The update equation used is:

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

fig J. Update equation for gradient descent. Source: AIMA (Ch. 18.6.4)

We chose to keep the default regularizer (L2) and default alpha hyperparameter (.0001), because we weren't sure how to select an optimal learning parameter.

Results

When we ran gradient descent for logistic regression, we were achieved our most best results with TF-IDF features, but extremely poor performance with raw counts (log losses were too high for the program to report). This certainly supports the use of TF-IDF to account for uneven term frequency.

Algorithm: Gradient Decent Log | Tokenization: Bag of words				
	Raw Count	Truncated Raw Count	TFIDF	Truncated TFIDF
Log loss	N/A	N/A	1.6569	1.7800
Accuracy	0.2599	0.2412	0.4279	0.4113

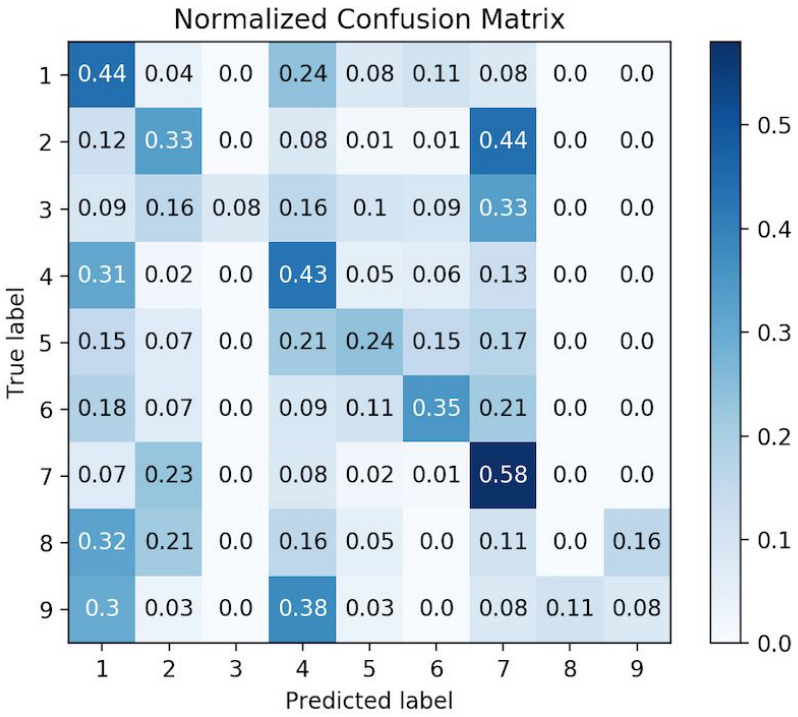

fig. K: Confusion Matrix for SDG logistic regression with TF-IDF features

Encouragingly, the classifier did a good job not only identifying common classes, but also the much smaller Class VIII and IX. We will revisit this algorithm in our "Analysis" section below.

Stochastic Gradient Descent with Modified Huber

In an early step, we used a stochastic gradient descent algorithm with a "Modified Huber" loss function, because it would return a probability that was useful for evaluating classification. The key to this algorithm is that it has a smoothed hinge loss. This means that it covers more outliers then a normal loss function and has a wider frame of reference. However, we are still unfamiliar with the loss function, and its poor performance led us to move on.

Results

Algorithm: Gradient Decent Modified-Huber | Tokenization: Bag of words				
	Raw Count	Truncated Raw Count	TFIDF	Truncated TFIDF
Log loss	11.6409	18.1123	15.6825	22.1590
Accuracy	0.2213	0.2644	0.2984	0.2002

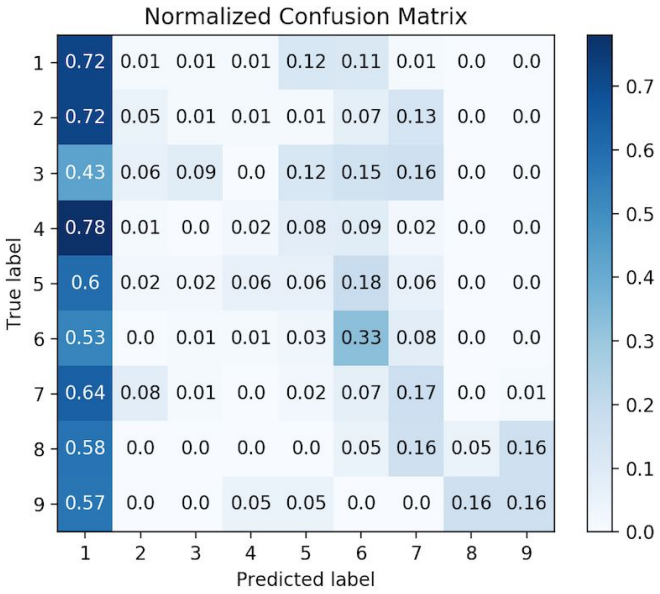

fig. L: Confusion Matrix for SDG Modified Huber with TF-IDF features

Random Forest

We finally used a random forest classifier, which we learned from other users on Kaggle. Although this was a new method for us, we understand that it fits decision tree classifiers on sub-samples of the dataset.[20] By randomly fitting decision trees and averaging results, it can avoid overfitting. We ran our decision trees to a depth of five in order to get results reasonably quickly.

[20] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Results

Random forest performed consistently (poorly) no matter what features were given, suffering the same fate as other regressions by guessing Class VII (most common) for most samples. We believe its consistent performance is due to its random nature.

Algorithm: Random Forest Classifier | Tokenization: Bag of words				
	Raw Count	Truncated Raw Count	TFIDF	Truncated TFIDF
Log loss	1.7172	1.7008	1.7095	1.6780
Accuracy	0.3604	0.3577	0.3643	0.3565

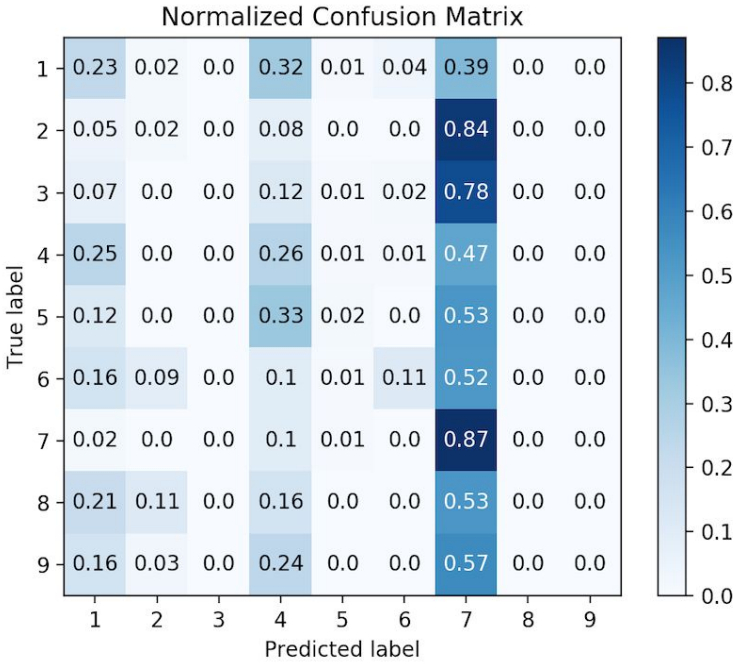

fig. M: Confusion Matrix for Random Forest with Truncated TF-IDF features

Analysis of Results & Areas for Improvement

As described above, Truncated TF-IDF produced the best feature data, and our Logistic Regression and Stochastic Gradient Descent algorithms returned similar (poor) accuracies on that data--about 40%, with log loss of 1.6 to 1.7. We believe the **Stochastic Gradient Model** (results below) actually performed better, as it seemed to make reasonable guesses about most classifications, whereas Logistic Regression achieved its score by being very confident that every sample was in Class VII, the most common class.

By comparison, the best-performing model on Kaggle achieved a log loss of .01909.

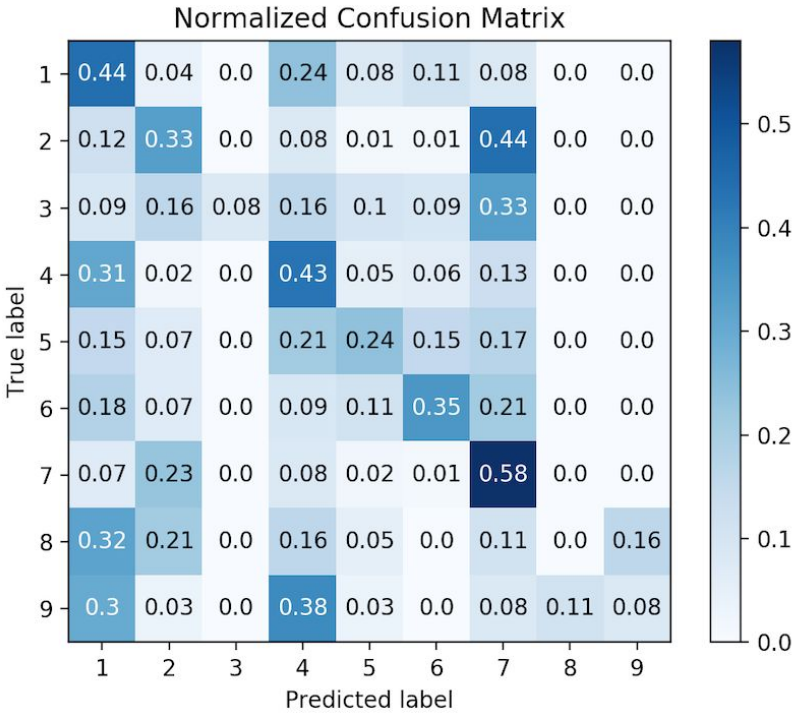

fig. N: Results for Stochastic Gradient Descent (LogLoss) and Truncated TF-IDF.

We believe that feature generation is probably holding us back. Medical text is hard to parse, and the same term frequency approach that works for classifying Yelp reviews fails us in this more complicated domain. One approach that we attempted, but were unable to get working in time, was a medical text parsing tool called GENiA tagger.[21] It was developed for reading MEDLINE abstracts, and is recommended for preprocessing biomedical data.

Other Kaggle users reported good results by using neural networks, "Gradient Boosting Machines", and "Generalized Linear Models". We will have to leave those solutions for another time!

Finally, a note that medical literature is consistently being updated. It would be helpful to provide some weight to more recent articles that revise our prior beliefs. However, time series regressions are currently beyond our ability.

[21] http://www.nactem.ac.uk/GENIA/tagger/

Appendix

We implemented our methods in python with the help of various libraries, particularly scikit-learn.[22] Our code and results can be found on our public GitHub repository, located here. The most important files are:

- MLEvaluator.py - Our working pipeline function, which can be configured to extract features in several ways, run several algorithms, and evaluate their performance.
- RESULTS for Final AI Project - A collection of supplemental charts and tables showing the performance of each (feature, algorithm) pair.

[22] In particular, we relied on the evaluation function provided in this kernel: https://www.kaggle.com/reiinakano/basic-nlp-bag-of-words-tf-idf-word2vec-lstm