Capstone Project
Machine Learning Engineer Nanodegree

Aaron Marquez
September 6, 2017

I. Definition

Project Overview

Personalized medicine is a medical procedure that divides patients into different groups with medical decisions, medicines, and products being tailored specifically to a patient based on their predicted response or risk of disease [1]. Also known as genome-based medicine, it is based on studying the human genome, looking at a person's genetic predisposition to certain medical conditions, and creating medicine designed for that person. It is a young but rapidly advancing field of healthcare and has a lot of potential to change how doctors treat illnesses and how medicine is prescribed. Instead of a single pill being designed for millions of people to consume as treatment for a disease, each person could have their own unique pill based on their genetic makeup.

Cancer research has learned a lot about genetic variety in cancer types and has created a new field called "Oncogenomics" that is the application of genomics and personalized medicine to cancer research and treatment. It has found success in targeted cancer treatments such as Gleevec, Herceptin, and Avastin. Machine learning can accelerate new drug discovery by combing through large amounts of data faster than a human could. A paper titled "Applications of Machine Learning in Cancer Prediction and Prognosis" talks about personalized medicine and the various ways machine learning can aid in cancer research [2].

Project Statement

Kaggle is currently hosting a competition titled "Personalized Medicine: Redefining Cancer Treatment" sponsored by Memorial Sloan Kettering Cancer Center (MSKCC). The following description is taken form the competition's homepage:

"Once sequenced, a cancer tumor can have thousands of genetic mutations. But the challenge is distinguishing the mutations that contribute to tumor growth (drivers) from the neutral mutations (passengers).
Currently this interpretation of genetic mutations is being done manually. This is a very time-consuming task where a clinical pathologist has to manually review and classify every single genetic mutation based on evidence from text-based clinical literature.
For this competition MSKCC is making available an expert-annotated knowledge base where world-class researchers and oncologists have manually annotated thousands of mutations.
We need your help to develop a Machine Learning algorithm that, using this knowledge base as a baseline, automatically classifies genetic variations [3]."

The purpose of this project is to create a machine learning model that uses natural language processing (NLP) to pore over thousands of medical literature text to determine the "class" of genetic mutation the text is describing. The classes in the datasets are numerical (1-9) but actually represent the different levels of whether a mutation is a driver or passenger.

Metrics

The official metric Kaggle uses is a multi class log loss formula. It is described by the following equation:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j})$$

Where N is the number of observations, M is the number of class labels, log is the natural logarithm, $y_{i,j}$ is the 1 if the observation i is in class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i is in class j. A lower log loss means better performance. This metric has been used in a few other Kaggle competitions as well.

I also used sci-kit learn's metric functions including log_loss, accuracy_score, classification_report, and confusion_matrix. Log loss uses the same equation as above to calculate a log loss score. Accuracy score calculates the accuracy and is defined as:

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$

Tp is true negative, *tp* is true negative, *fp* is false positive, and *fn* is false negative. Classification report builds a text report using the main classification metrics such as precision, recall, F1 score, and support. Precision, recall and F1 score are defined as:

$$Precision = \frac{tp}{tp + fp}$$

$$Recall = \frac{tp}{tp + fn}$$

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Support displays the number of times an observation is classified as a certain class. Finally, confusion_matrix is used to evaluate the accuracy of a classification. A confusion matrix is a num_classes by num_classes matrix where "each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa) [4]." All of these metrics are good to use for analyzing the performance of a classifier.

II. Analysis

Data Exploration

The data is split into training and testing sets. However the testing set that Kaggle provides does not include labels so to use the sci-kit learn metrics for testing the best thing to do is split the

training data (which does include labels) into new train/test datasets. The training data is in a file called “training_text.txt” and the labels are in a file called “training_variants.txt.” Here is a description of the files taken from Kaggle [5]:

- training_variants - a comma separated file containing the description of the genetic mutations used for training. Fields are ID (the id of the row used to link the mutation to the clinical evidence), Gene (the gene where this genetic mutation is located), Variation (the aminoacid change for this mutations), Class (1-9 the class this genetic mutation has been classified on)
- training_text - a double pipe (||) delimited file that contains the clinical evidence (text) used to classify genetic mutations. Fields are ID (the id of the row used to link the clinical evidence to the genetic mutation), Text (the clinical evidence used to classify the genetic mutation)
- test_variants - a comma separated file containing the description of the genetic mutations used for training. Fields are ID (the id of the row used to link the mutation to the clinical evidence), Gene (the gene where this genetic mutation is located), Variation (the aminoacid change for this mutations)
- test_text - a double pipe (||) delimited file that contains the clinical evidence (text) used to classify genetic mutations. Fields are ID (the id of the row used to link the clinical evidence to the genetic mutation), Text (the clinical evidence used to classify the genetic mutation)

There are a total of 3321 examples in the training dataset and 5668 examples in the testing set. I loaded the files into pandas dataframes and looked at the first few observations:

ID	Text
0	Cyclin-dependent kinases (CDKs) regulate a var...
1	Abstract Background Non-small cell lung canc...
2	Abstract Background Non-small cell lung canc...
3	Recent evidence has demonstrated that acquired...
4	Oncogenic mutations in the monomeric Casitas B...

Figure 1: First 5 observations from training_text.txt

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

Figure 2: First 5 observations from training_variants.txt

The column "Class" is what I am trying to learn and eventually predict but as you can see from figure 2, there are a couple extra columns called "Gene" and "Variation." According to some discussion in the Kaggle forums these are intended to be part of our input features. Any given gene/variation combination is unique in each dataset and points to a specific genomic coordinate. As an example, take the gene/variation pair with the ID 4 from figure 2. The gene CBL has a variation where L (Leucine) is changed to V (Valine) 399 places from the beginning of the CBL gene. These can be helpful features to feed to a model.

Exploratory Visualization

Next I thought I would be useful to see the distribution of the training dataset, specifically how many instances of each class appear. Using seaborn I plotted the following bar plot:

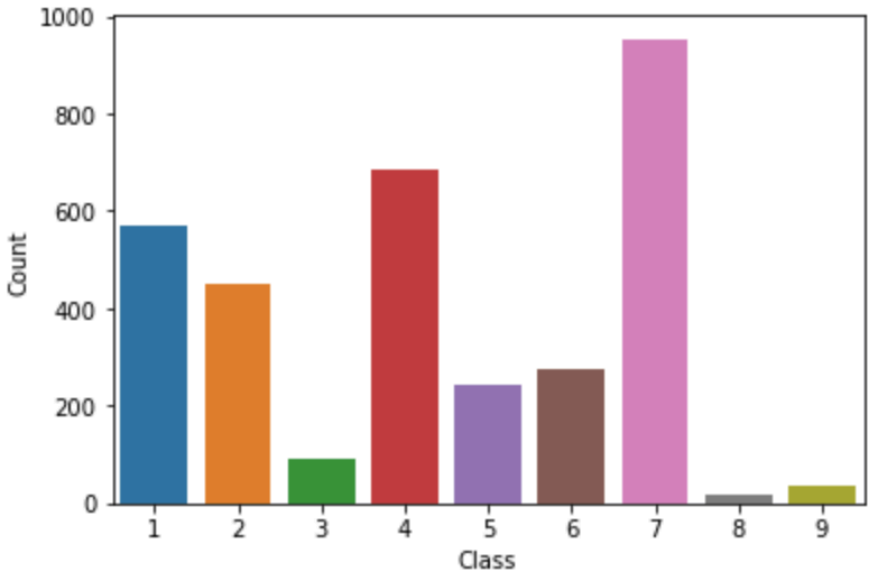

Figure 3: Bar plot of frequency for each class in training dataset

Figure 3 shows that there aren't many occurrences of the class 8 or 9 in the training dataset. Here is a description of the class distribution:

Class	Count
7	953
4	686
1	568
2	452
6	275
5	242
3	89
9	37
8	19

This uneven distribution of classes may have a negative effect on the model. Since it doesn't see very many examples of text with a class label of 8 or 9 then it might not be able to accurately classify these classes on the test set. One way of dealing with this problem is to perform over-sampling on the training dataset where more instances in class 8 or 9 can be created. SMOTE (synthetic minority over-sampling technique) is one method that does this [6]. Another option is to use decision tree classifiers such as random forest. Decision trees are known to work well on imbalanced datasets.

I also thought it might be interesting to see the distribution of frequently occurring genes in the variants training text file. In total there were 264 unique genes in the training dataset. Because of this I decided to plot the 20 most common genes using seaborn.

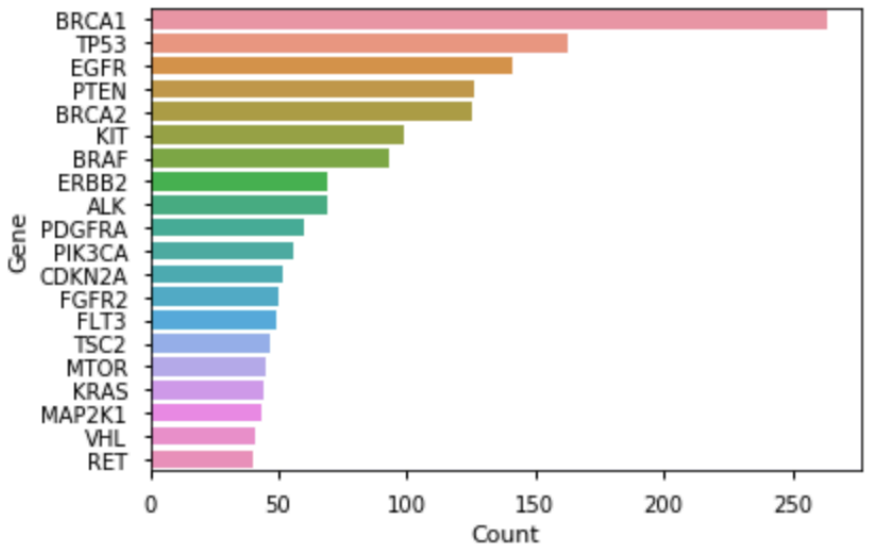

Figure 4: Frequency of 20 most common genes in training dataset

According to figure 4 BRCA1 is the most common gene in the dataset, appearing more than 250 times. This may be useful especially since Kaggle and the team at MSKCC meant for people to use the gene and variation columns as input features to our models.

Algorithms and Techniques

I originally planned to take two main approaches to this problem but ended up taking four main approaches in an attempt to get better results. The first is using sci-kit learn feature extraction methods to process the text into numerical data and using out of the bag classifiers from sci-kit learn. For my first approach I will use CountVectorizer and TfidfTransformer to process the text and then use a naïve bayes and decision tree classifier. GridSearchCV will also be used to find optimal hyper-parameters. The two main classifiers are MultinomialNB and RandomForestClassifier. In my capstone proposal I said I was going to use a GaussianNB classifier for my first approach, but after starting my project and doing more research on NLP I found that MultinomialNB classifiers give better performance than GaussianNB for nlp problems. It was a small change and I thought it would give me a better log loss for my out of the bag classifiers.

The second approach involves using spaCy to preprocess the text and prepare the data for deep learning. SpaCy is an industrial-strength natural language processing library for python [7] and keras is a deep learning python library [8]. Various neural network architectures will be used to get the best log loss score. Convolutional neural networks, multi-layer perceptron and recurrent neural networks are used.

The third attempt used gensim's Doc2Vec to create word embeddings and connect it to a neural network for deep learning. Gensim [9] is a python library used for NLP. I used similar deep learning architectures from the previous attempt.

Finally the fourth approach used pre-trained word embeddings, specifically pre-trained GloVe [10] (global vectors for word representation) vectors. Like the last two approaches, the word embeddings are connected to a keras neural network.

Benchmark

As a benchmark, I am using the log loss score of the team in the middle of the leaderboard. The leaderboard on Kaggle lists every team according to their log loss score. As of August 28th, 2017 there were 878 teams competing in the competition. The midmost team has a score of 0.74399 so one of my goals for this project is to achieve a better log loss score than that. If I am able to accomplish that then I will have done better than half of the current competitors. As a side note, it is unclear what methods that person used to obtain that log loss score because only the submission csv file is uploaded, not any code. But it should serve as an appropriate benchmark to surpass.

III. Methodology

Data Preprocessing

I started with my first approach and began using out of the bag sci-kit learn models, specifically MultinomialNB and RandomForestClassifier. I used CountVectorizer with default parameters to fit and transform the text data. CountVectorizer creates a sparse matrix of token counts and the number of features is equal to the vocabulary size found by analyzing the training data [11]. Then the newly created sparse matrix is transformed using TfidfTransformer (with default parameters). TfidfTransformer takes the sparse matrix as input and returns a normalized term-frequency times inverse document-frequency matrix [12]. The purpose of this is to give less weight to words that appear frequently in the text and may not be informative. For example if the word "the" appears a lot in the text then that word will be assigned a smaller weight because it appears often and isn't as informative as a word that appears less frequently such as "mutation."

In the exploratory visualization section I mention how there is a class imbalance in the training set. There are a low number of texts with the class 8 and 9. To solve this I used a python library called imbalanced-learn [13]. Within this library I used SMOTE (synthetic minority over-sampling technique) to make the class distribution more even. It was effective in creating more examples with the class 8 and 9.

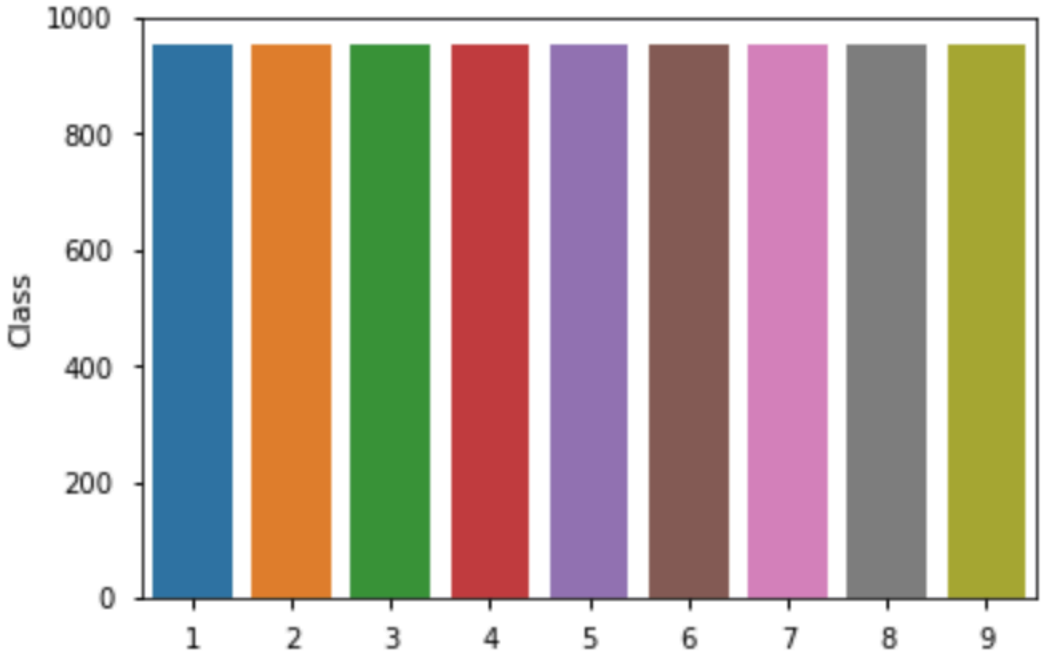

Figure 5: Bar plot of class distribution after using SMOTE

After resampling the data I also sci-kit learn's train_test_split method on the training data provided by Kaggle to create a new training/testing dataset. I did this because the testing set doesn't come with labels so to use any of sci-kit learn's metric methods I needed to create my own test set with labels. When I submit a final version to Kaggle however I train a model on the entire testing set provided by Kaggle, evaluate it on the testing set and submit a csv file with the predicted probabilities for each observation in the testing set. I made the test size be 20% of the total data and give it a random state for reproducibility.

For approach two I followed different pre processing steps. First I cleaned up the training text by removing stopwords, punctuation, and making all the words lowercase. Then I fed the cleaned

text into a spaCy parser object and grabbed the vector for each document. SpaCy's English model has built-in vectors for one million vocabulary words. It uses the 300-dimensional vectors trained on the Common Crawl corpus using the GloVe algorithm. With the word vectors for each document arranged in a matrix I one-hot encoded the training labels and was ready resample the data and feed it to a keras neural network.

Approach three followed a similar path as to approach two with one exception. Instead of using pre-trained word vectors I used Doc2Vec to create my own word embeddings based on the training and testing texts. Word embeddings is a technique meant to map semantic meaning into a geometric space. Similar words will have similar embeddings and this makes it easier to represent text as numbers without losing too much context. I also one-hot encoded the gene and variation features from the training and testing variants files and used TruncatedSVD to reduce the dimensionality of each to use as input, along with the word embeddings, to a neural network.

And finally for approach four the pre-trained GloVe vectors were used to compute the word embeddings, specifically the 100-dimensional embeddings of 400,000 words computed on a 2014 dump of English Wikipedia. This was then connected to a keras neural network.

Implementation

The first classifier used is a MultinomialNB (multinomial naïve bayes) classifier with default parameter settings. I chose multinomial naïve bayes because it is known to work well on text classification problems (as opposed to gaussian naïve bayes) and is typically used for discrete counts. After completing the steps described in data preprocessing I fit the MultinomialNB to the output of the TfidfTransformer and the training labels, transformed the test text using CountVectorizer and TfidfTransformer and called the classifiers predict function on the test text. I predicted the labels (to calculate accuracy), the log probabilities (to calculate the log loss score) and the probability estimates (for submitting final csv file to Kaggle). I also used sci-kit learn's Pipeline method to simplify the process of feeding the text to CountVectorizer, TfidfTransformer and the classifier.

I then followed the same process but replaced MultinomialNB with a RandomForestClassifier. I did this to get another set of predictions with an out of the bag classifier and I chose random forest because of its popularity with NLP problems.

Figure 6: Block diagram for MultinomialNB and RandomForestClassifier

In approach two I used spaCy for text preprocessing and followed the steps described in the previous section. After creating a matrix of word embeddings I created a several neural network models and evaluated the datasets on each, recording their log loss scores along the way. Keras was used to create the models. The first architecture was a multi-layer perceptron (MLP) with four hidden layers each with 512 neurons and using a relu activation function. For each hidden layer the weights were initialized with 'he_normal' [14][15]. The output layer had a softmax

activation function in order to get probabilities for each class when predicting. The next architecture I used was a one-dimensional convolutional neural network (CNN). There was the input layer, then a convolutional layer, a max pooling layer (pool size of 2), another convolutional layer, a global max pooling layer and a densely connected layer with softmax activation function. The first convolutional layer had 64 filters, size of 3, and a stride of 1. The second one had 128 filters and the same size and stride. Both had relu activation function and "same" padding, meaning the vector was padded with zeros so the filter doesn't go further than the end of a sequence (word vector). It essentially makes the output have the same length as the input. The third architecture was a recurrent neural network (RNN), specifically a bidirectional lstm (long short-term memory). There was only one lstm layer with 64 memory cells, a dropout layer (20% dropout rate) and a flatten layer, as well as the output layer with softmax activation function. Each architecture had stochastic gradient descent with nesterov momentum was used as an optimizer.

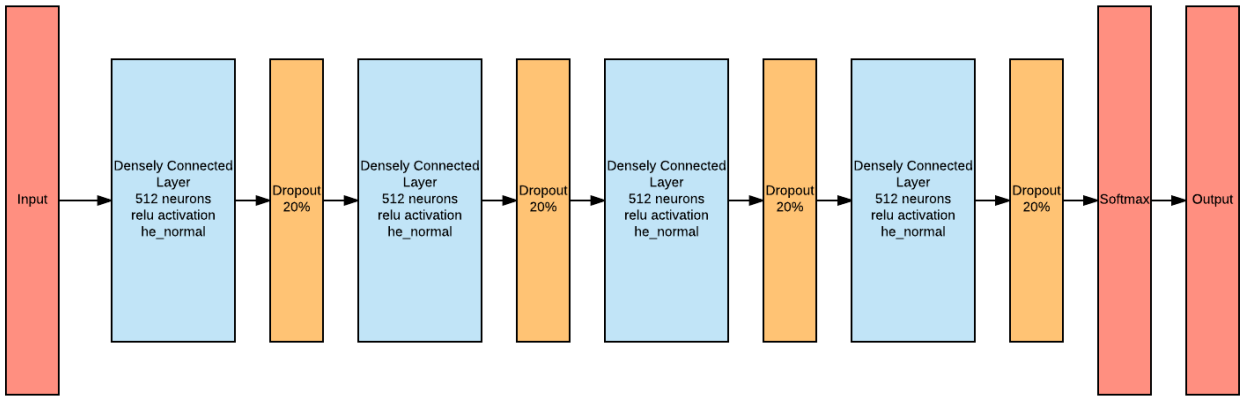

Figure 7: Basic MLP architecture

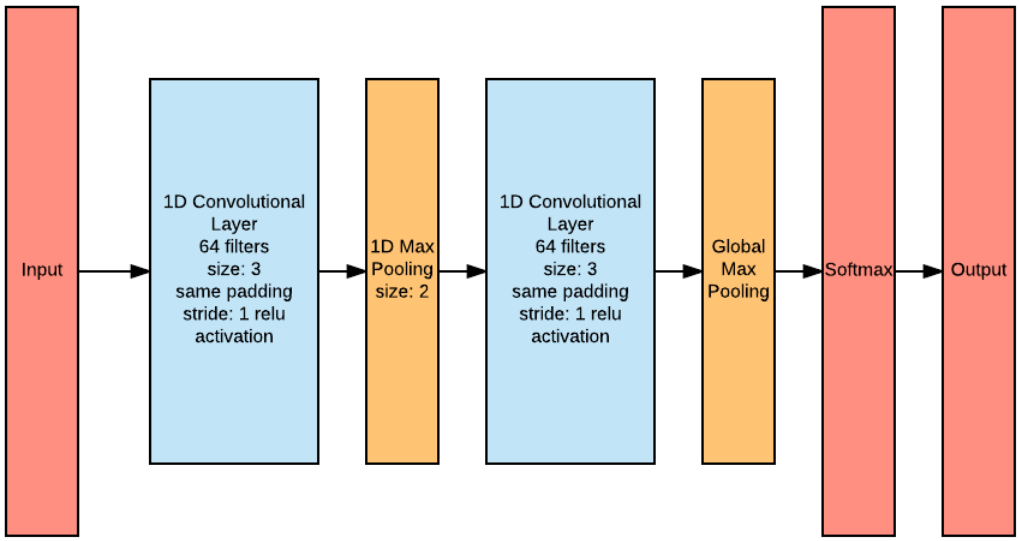

Figure 8: Basic CNN architecture

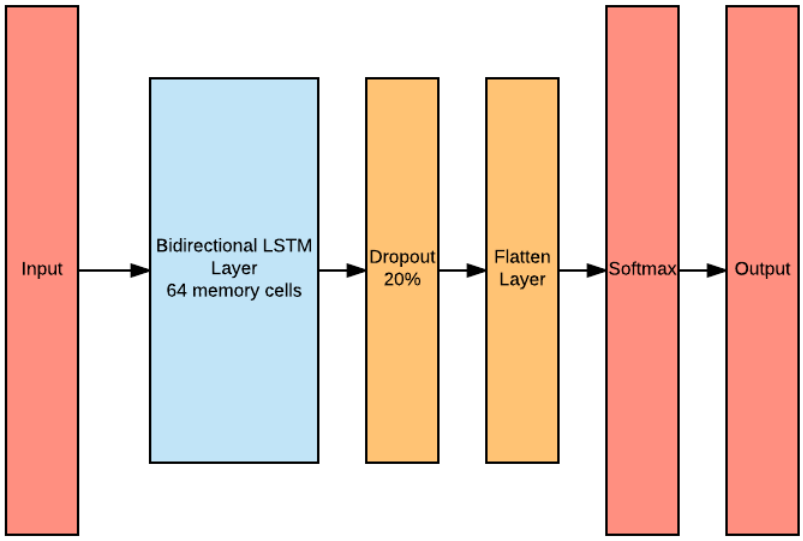

Figure 9: Basic LSTM architecture

The third approach used a custom spacy tokenizer to clean the text.

```
# Custom tokenizer
# Tokenizer parses through each document and removes stopwords, punctuation and personal pronouns
def spacy_tokenizer(sentence):
    tokens = nlp(sentence, parse=False, entity=False)
    tokens = [tok.lemma_.strip().lower() if tok.lemma_ != '-PRON-' else tok.lower_ for tok in tokens]
    tokens = [tok for tok in tokens if (tok not in stopwords and tok not in punctuations)]
    return tokens

vectorizer = CountVectorizer(analyzer='word', tokenizer=spacy_tokenizer)
```

Figure 10: Custom spaCy tokenizer

The custom tokenizer passes each document through spaCy, grabs the lemma (lemmatized version of a word), makes the word lowercase and removes stopwords and punctuation. The function called “spacy_tokenizer” is passed to CountVectorizer and used to tokenize all the text in the training and testing sets. It is important to note that I am tokenizing all the text from Kaggle’s training and testing sets. Then a Doc2Vec model is instantiated to create 300-dimensional word embeddings from the tokenized text. The one-hot encoded “gene” and “variation” features (with reduced dimensions) are concatenated to the word embeddings and then resampled using imbalanced-learn and fed into a keras model. The same neural network architectures used in approach two were used with some modifications I detail in the next section. Whenever I fit a keras model with a training set I also set the “validation_split” parameter to 0.2. This would take 20% of the training data and make it a validation set to test against. If the validation loss went down then the weights were saved to the model. Only the weights associated with the best validation loss were kept in the model.

The fourth and final approach used pre-trained GloVe vectors, similar to how spaCy uses them but the word vectors were grabbed directly from a text file the official Stanford GloVe website. The text is cleaned exactly the same way as in attempt three using “spacy_tokenizer” and transformed to sequences using keras text preprocessing methods. The word vectors are loaded from a text file (2014 English Wikipedia dump) and used to create a 100-dimensional embedding

matrix. This is then connected to a keras model using an embedding layer. The embedding matrix is acting as the weights for the embedding layer so that layer is not trainable. Two different CNN's were used for this approach.

When using an LSTM or CNN architecture I had to expand the dimensions of the training data. It was originally two dimensions (number of observations by size of embedding vector) but CNNs and LSTMs require three dimensions as input. So I used numpy to expand the dimensions (number of observations by size of embedding vector by 1). Earlier it is mentioned that the Kaggle testing set doesn't come with labels so the only way to get a log loss score is to make prediction on the test set and submit it as a csv file to Kaggle. But to make things easier and fast to test I split the training data into its own train/test set. This was done after creating the word embeddings. Now I could get a log loss score using sci-kit learn without having to submit to Kaggle.

Refinement

I made a lot of parameter changes to every model I used. Sometimes it didn't make a difference and sometimes it performed better. For approach one I used GridSearchCV to sift through a set of hyper-parameter values and find the optimal ones. I did this for the naïve bayes classifiers. I wanted to find the best possible log loss scores for at least one out of the bag classifier before moving on to use spaCy and keras. For MultinomialNB I used the following parameters for GridSearchCV:

CountVectorizer: ngram_range: (1,1), (1,2)
TfidfTransformer: use_idf: True, False
MultinomialNB: alpha: 0.1, 0.01

GridSearchCV tries all combinations of those parameters and chooses the ones that give the best performance.

For approaches two, three, and four most of the refinement was in tweaking the parameters of the neural networks. In approach two I used an MLP with four hidden layers, each with 512 neurons and relu activation, and four dropout layers with 20% dropout rate, and an output layer with softmax activation. For approach two I didn't modify the MLP at all. I spent most of the time modifying the CNN and LSTM. I did modify the MLPs used in approach three and four however. In approach three the MLP had four hidden layers. The first hidden layer had 1024 neurons, "normal" weight initialization, and relu activation. Second hidden layer was the same except it had 512 neurons, third layer had 256 neurons, and last hidden layer had 128 neurons. Each hidden layer (except the last) had a dropout layer. First dropout layer had 30% dropout rate and the second and third had 50% dropout rate. I evaluated a log loss three times for this architecture: one as is described above, one with "he_normal" initialization, and another time with extra neurons in the each hidden layer (twice the current amount). In approach four I made similar modifications to the MLP.

The CNN architectures had modifications made to the number of filters, filter size, stride, padding, and the number of convolutional layers. When training with CNN I noticed that the log loss was steadily going down after 25 epochs so I tried increasing the number of epochs to 100

and saw it go down further. This happened many times and sometimes for I had to decrease the number of epochs because it would start to overfit the training data and the validation loss would go up.

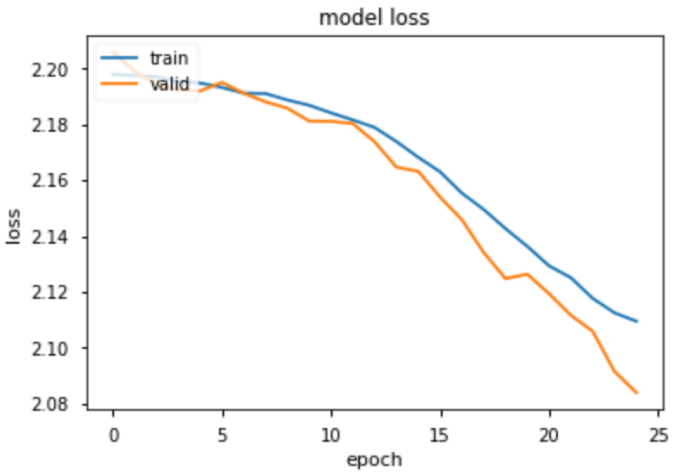

Figure 11: More epochs needed, log loss still decreasing

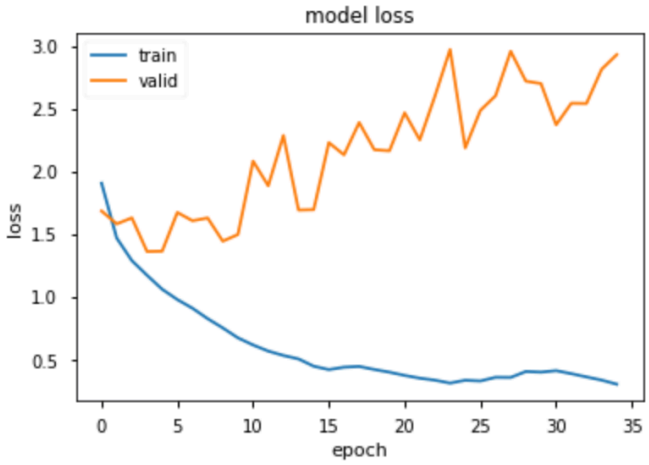

Figure 12: Too many epochs, model is overfitting

I had some difficulty making modification to the LSTM architecture. If I put too many memory cells (512 or 1024) then training would take too long or my computer would run out of memory and crash. Because of this I used only one bidirectional LSTM layer with 64 cells. I also used a dropout layer with 20% dropout rate and a flatten layer so the output would be a two dimensional matrix. The output, as always, was a dense layer with 9 neurons (because there are 9 classes) with softmax activation. When using Doc2Vec in approach three I also tried two bidirectional lstm layers. First layer had 128 cells and the second had 64 cells. I also tried changing the weight initializer, dropout rate, and with no bidirectional wrapper.

For approach four I changed some of the parameter settings in the embedding layer, such as making the weights trainable (originally the weights were taken from pre-trained GloVe vectors so it didn't need to be learned during training),

IV. Results

Model Evaluation

The model that gave the best performance was with approach three using spaCy for text preprocessing, Doc2Vec for word embeddings, and keras for neural networks. The model architecture I used was a single LSTM layer with 64 memory cells, 20% recurrent dropout rate, 20% dropout rate, and a "normal" weight initializer. Using this model I received a log loss score of 1.92953 from Kaggle. I think it is a good model to use for this specific problem. Because the data is medical text and contains a lot of medical terms that aren't common, it was good to use Doc2Vec to create word embeddings from the text itself instead of using pre-trained word embeddings from spaCy or GloVe. A validation set was always used during model training to

lower the chance of overfitting. Log loss is also a good metric to measure how good a model is. I noticed that sometimes the accuracy of naïve bayes or random forest classifiers would be high but I would get a poor log loss when submitting to Kaggle.

Justification

After making predictions (probabilities, log probabilities, and labels), I was able to calculate the log loss and accuracy as well as create a classification report detailing the precision, recall, F1-score, and confusion matrix. When I predicted the probabilities for each class I had to create a text file and write all the probabilities on each line with the observation ID too. This was the submission format when I was ready to submit my predictions to Kaggle. The format looked like this:

```
ID,class1,class2,class3,class4,class5,class6,class7,class8,class9
0,0.00,0.00,0.00,0.00,0.00,0.00,1.00,0.00,0.00
1,0.05,0.02,0.00,0.26,0.00,0.00,0.67,0.00,0.00
2,0.00,0.00,0.00,0.02,0.00,0.00,0.98,0.00,0.00
3,0.00,0.00,0.00,0.00,0.00,0.00,0.99,0.00,0.00
4,0.01,0.00,0.00,0.07,0.00,0.00,0.93,0.00,0.00
```

Figure 12: Submission file format

First attempt I submitted to Kaggle got a log loss score of 7.33243 and placed me in 883rd place out of 1029 competitors as of September 2nd. This was with the CountVectorizer, TfidfTransformer (default parameters), and MultinomialNB classifier. My second attempt I submitted to Kaggle using the RandomForestClassifier with default settings. I was able to achieve a log loss score of 2.98226 and advanced 14 places on the leaderboard (see below for a complete list of log loss scores). By comparison, for MultinomialNB, the log loss score I got using my own train/test set was 2.197 and the accuracy was 0.53. The reason for such a large difference between these two log loss scores might be because the score I got using my own train/test set was using imbalanced-learn and when I first submitted my first submission file I hadn't tried using imbalanced-learn yet. Kaggle is also evaluating predictions made on a different test set than I had used.

	precision	recall	f1-score	support
1	0.55	0.49	0.52	110
2	0.43	0.67	0.53	107
3	0.26	0.52	0.34	21
4	0.74	0.50	0.60	129
5	0.27	0.41	0.32	37
6	0.82	0.65	0.73	63
7	0.70	0.46	0.55	185
8	0.05	0.50	0.09	2
9	0.52	1.00	0.69	11
avg / total	0.61	0.53	0.55	665

Figure 14: Classification report for MultinomialNB

```
[[54  7  2 15 13  8  3  5  3]
 [ 3 72  2  1  2  0 23  4  0]
 [ 1  1 11  3  3  0  2  0  0]
 [31  5  7 65 13  0  1  3  4]
 [ 5  7  4  1 15  1  3  0  1]
 [ 2  6  0  1  7 41  5  1  0]
 [ 3 68 17  2  3  0 85  6  1]
 [ 0  0  0  0  0  0  0  1  1]
 [ 0  0  0  0  0  0  0  0 11]]
```

Figure 15: Confusion matrix for MultinomialNB

According to the classification report above, recall and precision scored close to each other and the average F1 score was 0.55, which is a mediocre score (a better F1 score is closer to one than

zero). The individual F1 scores for each class also tell a different story. Class 3 and class 8 received not so good F1-scores. Class 8 got 0.09 meaning hardly any observations were correctly classified as being class 8. The highest F1 score was 0.73 for class 6. The confusion matrix is organized so that the rows are the actual class labels and the columns are the predicted labels. So the diagonal numbers are correct classifications. The classifier predicted the label 2 68 times when the actual label was 7. It also had some trouble labeling class 1. It got it right 54 times but confused it with class 4 31 times.
The best parameters GridSearchCV found for MultinomialNB was:

CountVectorizer: ngram_range: (1,1)
TfidfTransformer: use_idf: True
MultinomialNB: alpha: 0.01

Using spaCy and keras I was able to get better log loss scores on Kaggle. Below are some of the many log loss score predicted using sci-kit learn's log loss method (using training/testing set made from Kaggle's training set) and Kaggle's own evaluation.

- Approach 1 (MultinomialNB and RandomForestClassifier)
  - MultinomialNB
    - Log loss (sci-kit learn, default parameters): 2.1972
    - Log loss (sci-kit learn, GridSearchCV): 2.19722
    - Log loss (Kaggle, GridSearchCV): 5.09670
    - Log loss (Kaggle, default parameters): 7.33243
  - RandomForestClassifier
    - Log loss (sci-kit learn, default parameters): 1.9372
    - Log loss (Kaggle, default parameters): 2.98226
- Approach 2 (spaCy and keras)
  - MLP
    - Log loss (sci-kit learn, 25 epochs): 1.397288
    - Log loss (Kaggle): 2.47576
  - CNN
    - Log loss (sci-kit learn, 25 epochs): 2.10795
    - Log loss (sci-kit learn, 100 epochs): 1.44532
    - Log loss (Kaggle): 2.96044
  - Bidirectional LSTM
    - Log loss (sci-kit learn, 25 epochs): 1.77028
    - Log loss (Kaggle): 2.04779
- Approach 3 (spaCy, Doc2Vec, keras)
  - MLP
    - Log loss (sci-kit learn): 0.65002
    - Log loss (Kaggle): 19.45819
  - CNN
    - Log loss (sci-kit learn): 1.86835
    - Log loss (Kaggle): 1.95406
  - Bidirectional LSTM
    - Log loss (sci-kit learn, 2-layers): 1.8187873
    - Log loss (Kaggle, 2-layers): 1.93370

        - Log loss (Kaggle, 1-layer LSTM, no Bidirectional wrapper): 1.92953
- Approach 4 (spaCy, GloVe, keras)
    - MLP
        - Log loss (sci-kit learn): 3.01928
        - Log loss (Kaggle): 7.54333
    - CNN
        - Log loss (sci-kit learn): 3.01928
        - Log loss (Kaggle): 7.54333

In my proposal I put as a benchmark to get a better log loss score than the person in the middle of the leaderboard, which was 0.74399. I did not get a better log loss score than that at any time during the competition. But as of October 2nd, 2017 the competition has now ended and I am in 357th place on the public leaderboards out of 1386 competitors, and on the private leaderboards I am in 129th place. Kaggle says that they use 76% of the test data for calculating leaderboard position. There is a private leaderboard that is based on the other 24% of the data and won't be viewable until the competition ends. So I did end up placing higher than at least half the competitors.

V. Conclusion

Free-Form Visualization

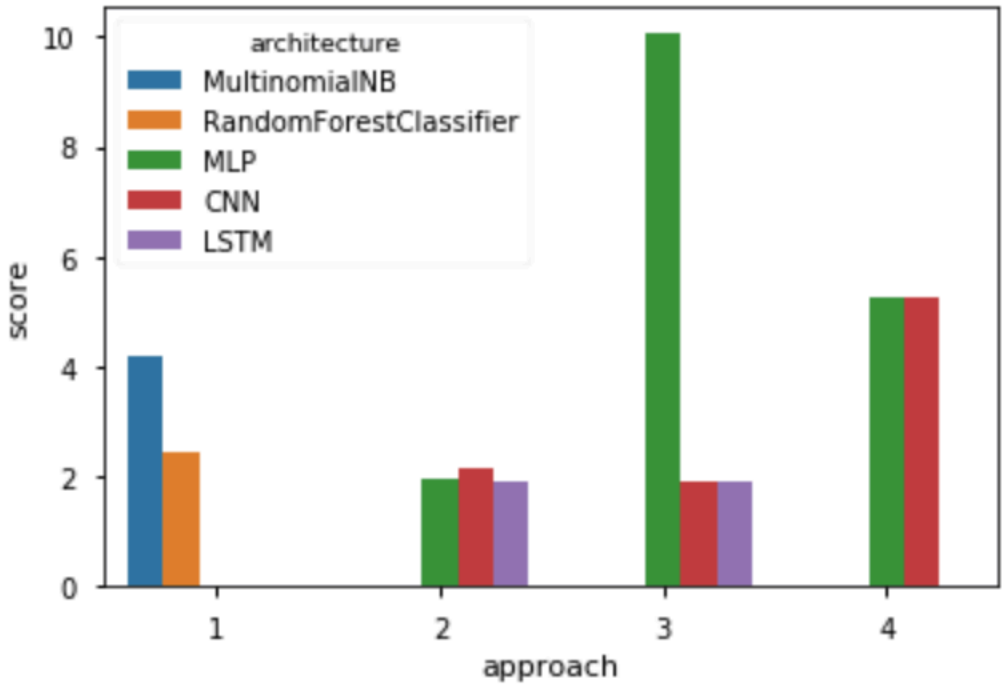

Figure 16: Bar plot of log loss scores

Figure 16 shows the log loss score across all approaches taken, including score form sci-kit learn and Kaggle. Approach one gave a high log loss for MultinomialNB but RandomForestClassifier did a little better. Approach two had low log losses for all three neural network architectures, but it was approach three that gave the lowest score with LSTM even though it also gave the highest score as well with MLP architecture. Approach four gave high log losses for both MLP and CNN architectures.

Reflection

At the beginning of this project I had no experience in natural language processing and knew that if I would have to do a lot of reading to do an adequate job. I ended up learning a lot about NLP and other things that surprised me as well such as dealing with imbalanced data and working with seaborn for visualization. I read about different techniques for dealing with that problem and came across the imbalanced-learn python library, which was extremely useful. I also learned a lot about text preprocessing with spaCy and recurrent neural networks and their uses for NLP problems. It was also helpful to read discussions on the competition's Kaggle forum, especially from people who organized the competition from MSKCC and the experts who know a lot about oncology and cancer research. I first thought that I would get terrible scores with naïve bayes or random forest but it did better than I though especially random forest. When spaCy and keras together weren't getting any better with the modifications I was making I wondered if because the text had a lot of medical terminology in it that the model had a hard time of modeling those terms. I also read some people using Word2Vec and Doc2Vec in the competition and decided to use it since it made more sense to train my own word embeddings from the provided text instead of using pre-trained ones. It was difficult to tweak the parameters for the neural networks because there were so many things that could change (number of layers, neurons, activation, filters, size, stride, etc.) and it wasn't always clear what worked and why. On top of that training would sometimes take long especially with LSTM architectures. But after reading about a lot of articles and tutorials on NLP and RNN's it became easier to work with them.

Improvement

Something interesting I noticed was that after cleaning the text, words that had a hyphen in them like "cyclin-dependent" became "cyclin" and "dependent." I wondered if this made a difference when creating the word embeddings and maybe they should be stuck together like "cyclindependent" instead. It might be good to try and train a new model with that change and compare the log losses. I used 300-dimensional vectors for Doc2Vec but this can be changed. Trying other dimensions such as 300, 400, 500, etc. might be useful. Also spending more time tweaking the neural networks could lead to better results. Sometimes my computer ran out of memory and crashed if I used too many memory cells in an LSTM layer, so training on a more powerful GPU could also lead to faster training times and make it easier to try deeper neural networks.

There is also the possibility of using more input features than just text, gene, and variation. I read in the forums that there is a lot of external data that can useful for this problem. Not all external data was allowed but it might yield a better score to incorporate external data into the model. Using random forest as part of an ensemble using gradient boosting (such as XGBoost) could also give better results. This was a cool project to work on is also a really interesting and important problem and NLP is still evolving and making advancements every year. It will be

interesting to see how machine learning is applied to more NLP problems and what methods and architectures come out on top.

References

[1] https://en.wikipedia.org/wiki/Personalized_medicine
[2] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2675494
[3] https://www.kaggle.com/c/msk-redefining-cancer-treatment
[4] https://en.wikipedia.org/wiki/Confusion_matrix
[5] https://www.kaggle.com/c/msk-redefining-cancer-treatment/data
[6] http://www.jair.org/papers/paper953.html
[7] https://spacy.io
[8] https://keras.io
[9] https://radimrehurek.com/gensim/index.html
[10] https://nlp.stanford.edu/projects/glove/
[11] http://scikit-learn.org/dev/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
[12] http://scikit-learn.org/dev/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html#sklearn.feature_extraction.text.TfidfTransformer
[13] http://contrib.scikit-learn.org/imbalanced-learn/stable/
[14] https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf
[15] https://keras.io/initializers/