Pathology Notes Project

Technical Report

Matthew Rossi

matthew.rossi@miami.edu

Spring 2022

Contents

1. Overview.

The Pathology Notes Project is a natural language processing (NLP) project that aims to build workflows to extract meaningful, useful data from unstructured cancer pathology reports pulled from the University of Miami's Jackson Memorial Hospital. Ultimately, such workflows could be deployed in a medical environment as an aid to doctors. Last semester's work focused primarily on data discovery and understanding; data preprocessing and cleaning; and building pipelines for classifying documents by diagnosis code as a proof of concept for other classification tasks. The work this Spring 2022 built upon the previous work, focusing on configuring and optimizing a few deep learning approaches for the data.

Research was funded by the University of Miami's Institute for Data Science and Computing (IDSC) and was conducted using the supercomputer TRITON, which is owned and operated by IDSC.

The bitbucket repository for this project is `/ccssweng/pathologynotes/`.

My work on this project was supervised by Amin Sarafraz (a.sarafraz@umiami.edu) and Chris Mader (cmader@med.miami.edu).

2. Environment configuration.

To execute the python code within the TRITON environment, it is necessary to configure a personal environment using conda that will contain all the relevant Python packages.

IBM provides Watson Machine Learning Community Edition (WML CE), which contains many commonly used deep learning packages. The following commands create a new environment with these packages installed:

```
ml wml_anaconda3

conda create -n nlp python=3.7 powerai=1.7.0

conda install -c conda-forge gensim
```

3. Job submission.

The TRITON environment uses a job scheduler[1] to allocate resources. The repository contains job scripts in the /jobs/ directory. Some scripts need to be edited prior to use. Usage is as follows:

A. Traditional machine learning models.

Open `trad_ml.job` and edit the python call line to refer to the desired dataset file.

Then submit the job using:

- `bsub < jobs/trad_ml.job`

B. Deep learning models.

First, open `feature_extraction.job` and edit the python call line to refer to the desired dataset file.

Then submit the job using:

- `bsub < jobs/feature_extraction.job`

Then run the desired architecture using one of:

- `bsub < jobs/cnn.job`
- `bsub < jobs/han.job`
- `bsub < jobs/hcan.job`
- `bsub < jobs/hisan.job`

[1] https://acs-docs.readthedocs.io/triton/3-jobs/README.html

4. Background.

This project was inspired by a series of research papers published by Shang Gao and others. In particular, the project aims to apply to a new dataset the deep learning architectures presented in the papers *Hierarchical Convolutional Attention Networks for Text Classification*[2] (2018) and *Classifying cancer pathology reports with hierarchical self-attention networks*[3] (2019). Both papers are concerned with text classification, an NLP task in which a sentence or document is provided as input and the task is to predict the correct label. The 2018 paper, for example, trains a neural network to predict the number of stars that accompany a Yelp review, while the 2019 paper trains a neural network to predict a variety of features of a tumor based on the text contained in a pathology report.

Machine learning (ML) approaches to natural language processing rely on first converting the categorical text data to numerical form via a word embedding process. Word embedding algorithms create a vector representation of a specified length (called the embedding dimension) of each word in the vocabulary of a corpus. Traditional ML techniques for classification rely on analyzing the relationship between vectors in the embedded vector space.

Term frequency-inverse document frequency[4] (TF-IDF) vectorization is a simple document embedding algorithm. It uses a simple assumption: the more often a word appears in a document (term frequency), the more relevant it is to the meaning of that document, unless the word appears frequently throughout the corpus (inverse document frequency). Term frequency of a term t is calculated as:

$$TF(t) = \frac{number\ of\ times\ a\ term\ appears\ in\ a\ document}{total\ number\ of\ terms\ in\ the\ document}$$

Inverse document frequency for a term t is calculated as:

$$IDF(t) = \ln\left(\frac{total\ number\ of\ documents}{number\ of\ documents\ with\ term\ t\ in\ it}\right)$$

The score for a term t in each document is the product of these values. As such, the output of TF-IDF vectorization is a sparse matrix where each row is a document, each column is one term in the vocabulary, and each element is the TF-IDF score of that term in that document. Each document vector has length equal to the size of the corpus vocabulary. However, since most words in the full vocabulary do not appear in any given document, most entries are zero, so the resulting matrix is sparse with shape (number of documents, vocabulary size). This sparse output can be fed to traditional machine learning algorithms.

[2] **Invalid source specified.**

[3] **Invalid source specified.**

[4] http://tfidf.com/

Word2Vec[5] is a more sophisticated word embedding algorithm that trains a neural network to predict a word based on the words in a small neighborhood around it (called context words). In completing this task, it generates vector embeddings (of a chosen, fixed embedding dimension) for each word in the vocabulary. It outputs a dense vocabulary matrix with shape (vocabulary size, embedding dimension).

The documents in the corpus can then be "translated" to numerical form, where each word in the document is replaced with an integer that refers to that word's row index in the vocabulary matrix. The vocabulary matrix and translated documents can be fed to deep learning architectures like those presented in the papers by Gao et al.

For this project, I tested four traditional machine learning classifiers: Multinomial Naïve Bayes, Logistic Regression, Random Forest Classifier, and Support Vector Classifier. The results from these classifiers serve as a baseline against which to compare deep learning methods.

These traditional ML methods have been surpassed by deep learning approaches like recurrent neural networks (RNNs), convolutional neural networks (CNNs), and the architectures introduced by Gao et al. in 2018 and 2019, among others. As the authors explain, these deep methods show better performance because the traditional methods rely on either hand-engineered features or the appearance of specific character or word sequences of a particular length (n-grams). When text data does not conform to pre-determined rules, these approaches fail. Deep learning methods, however, are more flexible and can learn features directly from the text without the need for extensive human feature engineering.

CNNs, which are traditionally used for computer vision tasks, have been applied to NLP tasks with some success. Such networks rely on "convolution layers" that utilize a set of sliding window "filters" of a set size that operate on only a few words at a time before sliding the window to operate on the next set of words. A CNN might be composed of a series of convolutional layers, each of which operates on the output of the previous layer. An accurate CNN works because, through the process of repeated convolutions, it distills semantic data from the complex text, which can be more easily classified.

In 2017, Vaswani[6] et al. introduced the Transformer, a deep learning model based entirely on self-attention mechanisms. As the authors explain, "An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key." Essentially, the query vectors specify where to look or what to look for in a sentence; the key vectors are used to as answers to those queries; and the value vectors are attached to the keys, so they are retrieved in the process. Self-

[5] https://radimrehurek.com/gensim/models/word2vec.html

[6] **Invalid source specified.**

attention is distinct from traditional attention because it generates queries, keys, and values all from the same sentence or document. Self-attention allows for the efficient comparison of all pairs of words in a sentence.

In the first paper, Gao et al. introduce the Hierarchical Convolutional Attention Network (HCAN), which combines the principles of convolutional neural networks with self-attention mechanisms. The convolutional multihead self-attention mechanisms they employ first apply convolutions for a given sentence to generate inputs to the self-attention heads, whose output is a sentence embedding. The process is repeated for the set of sentence embeddings to produce a document embedding; hence, the architecture is "hierarchical". The document embedding is then used for classification. The HCAN achieved state-of-the-art performance on four classification tasks: Yelp review sentiment, Amazon review sentiment, Amazon review product category, and Pubmed abstract topic.

In the 2019 paper, Gao et al. present a new deep learning architecture, the Hierarchical Self-Attention Network (HiSAN), which they specifically designed for the task of classifying cancer pathology reports, a task which has unique difficulties not found in most text classification tasks. They train the HiSAN to predict classes for five key cancer data elements – site, laterality, behavior, histology, and grade – using a dataset of approximately 375k labeled cancer pathology reports.

The HiSAN is similar to the HCAN; it combines the convolutional multihead self-attention mechanisms present in the HCAN with a multihead target attention, which compares each entry in the convolutional multihead self-attention output sequence S with a learned target vector that represents what information the model should be looking for to perform the current task.

The metrics used for evaluating the effectiveness of a neural network trained for classification generally fall into two categories: speed and accuracy. Training time is the amount of time spent running training examples through the network to tune its parameters to achieve a satisfactory model. Inference time is the amount of time required for a trained model to classify a given document. Accuracy is often measured as the number of correctly labeled datapoints divided by the total number of datapoints. However, such a metric is misleading for a dataset with imbalanced classes. For example, if a disease affects only 1% of a population, a model that predicts that no one has the disease attains 99% accuracy but is clearly a useless model.

For this reason, alternative performance metrics are commonly used. For a classifier:

$$Precision = \frac{number\ of\ items\ correctly\ predicted\ to\ be\ in\ a\ certain\ class}{total\ number\ of\ items\ predicted\ to\ be\ in\ that\ class}$$

$$Recall = \frac{number\ of\ items\ correctly\ predicted\ to\ be\ in\ a\ certain\ class}{total\ number\ of\ items\ that\ actually\ belong\ to\ that\ class}$$

F1 score in the harmonic mean of precision and recall. Its lowest value is 0 (if either precision or recall is 0), and its best value is 1 (indicating perfect precision and perfect recall). For a multiclass problem, the macro F1 score is the average F1 score across all classes.

5. Our dataset.

The dataset for this Pathology Notes Project is a .tsv file containing entries separated by tab stops. It has 197491 rows and three columns: 'c.reident_key' (a 12 digit patient ID), 'c.icd10_after_spilt' (an ICD-10 diagnosis code), 'c.path_notes' (the text of the pathology report). The patient IDs are discarded since they have no predictive power. The diagnosis codes are ICD-10 codes. ICD-10 is the current edition of the International Statistical Classification of Diseases and Related Health Problems, a medical classification list provided by the World Health Organization. These codes are used for insurance and billing purposes. As such, there's no guarantee that they're assigned by a doctor or nurse.

In the 197,491 datapoints in the set, there are 1394 unique diagnoses (including subclassification, so these look like AXX.YYY). Excluding the subclassifications (for example, considering only AXX) yields 141 top-level classifications.

This dataset contains only codes C00-D49, which are neoplasms, or tumors. Codes beginning with C denote malignant neoplasms, while those beginning with D denote neoplasms that are either benign or have unspecified or uncertain behavior. The two digits after the letter indicate the tumor's site and are accompanied by a subclassification that gives more information. For example, C50 is a malignant neoplasm of the *breast*; C50.1 is a malignant neoplasm of the *central portion* of the breast; C50.11 is a malignant neoplasm in the central portion of the *female* breast; and C50.112 is a malignant neoplasm in the central portion of the *left* female breast.

As such, by slicing a different location from the ICD-10 code, we can build datasets with different sets of class labels to train unique classifiers. For example, these datasets could train a classifier for a variety of tasks: predicting whether a tumor is malignant or benign; predicting the site of a tumor; predicting the subsite of a tumor; or predicting whether a patient is male or female.

The text data for each datapoint is a concatenation of the text of all pathology reports that have the same patient ID and ICD-10 code. It seems that each document has reports in chronological order, but dates are not attached in any structured way. Sometimes reports contain the year. Identifying information, such as name, medical record number (MRN), case number, date of birth, etc. has been redacted. The text data has little structure and lots of noise.

The pathology text of a single datapoint might look something like:

For context, this is less than 1/5th of the text for this datapoint. In this small sample, although there's clearly a lot of junk (such as the many '=' characters used to format report headings, or the redacted '[****]'), there are also many meaningful medical terms.

Although work thus far has focused on classification using the ICD-10 codes as ground truth labels, there is additional information contained in these reports that could be extracted and used for classifying other elements of the tumors, such as grade/stage or size. If an expert annotated the data with labels for these other features, running classification on this new set of labels would be trivially different from classification with the ICD-10 codes as targets.

6. Text cleaning and pre-processing.

Pipelines for data cleaning and pre-processing were designed and debugged in Jupyter interactive python notebooks.

Noisy data that contains junk is not conducive to machine learning methods for a few reasons. First, it increases the vocabulary size, which makes word embedding algorithms less effective. A model cannot learn the meaning of certain words when they are surrounded by meaningless tokens, and the resulting vector representation is thus less useful. Second, a longer document is a datapoint of higher dimensionality, which takes more time to analyze and is more difficult to accurately extract meaning from.

Thus, the text cleaning and pre-processing for this project takes a two-stage approach: 1. formatting using re and 2. stop words.

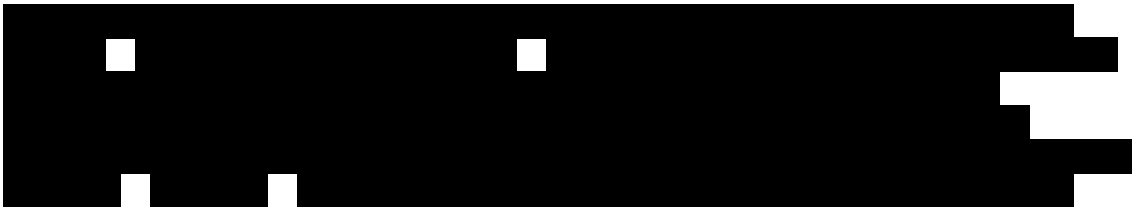
A. Formatting using re.

The **re** python package allows for the use of regular expressions to find and replace patterns in strings. The command `re.sub(pattern, repl, string)` replaces occurrences of `pattern` in the given `string` with the new string `repl`.

The following table explains the lines used for preprocessing, where `text` is a string that holds a single report:

`text = text.lower()`	Lowercase the text.
`text = re.sub("dr\.",'dr', text)`	Remove periods from "dr.", "m.d.", "a.m." and "p.m." so that these periods are not interpreted as the ends of sentences.
`text = re.sub('m\.d\.', 'md', text)`	
`text = re.sub('a\.m\.','am', text)`	
`text = re.sub('p\.m\.','pm', text)`	
`text = re.sub("\d+\.\d+", 'floattoken', text)`	Replace all decimal numbers with a token that represents a float number.
`text = re.sub("\.{2,}", '.', text)`	Replaces sequences of multiple periods with single periods.
`text = re.sub('[^\w_|\.|\?|!]+', ' ', text)`	Removes any characters that are neither alphanumeric characters (\w) nor are the punctuation marks ".", "?", "!" (i.e., removes symbols).
`text = re.sub('\.', ' . ', text)`	Adds space around punctuation marks so that the end of sentences can be identified.
`text = re.sub('\?', ' ? ', text)`	
`text = re.sub('!', ' ! ', text)`	
`text = re.sub('\d{3,}', '', text)`	Removes all occurrences of 3 or more straight numeric characters.

After being processed, the example text above would look like:



The function for text pre-processing is contained in
`/pathologynotes/utilities/process_text.py`

B. Stop words.

A stop list contains stop words, which are words carefully selected to be filtered out of a text before applying natural language processing methods. In the context of this project, there are two types of stop words worth removing because they have no predictive power for the models. The first set is those words that are extremely common in the English language and therefore have little to do with the meaning of a sentence, such as "a", "and", "the", "but", simple prepositions, and more. The second set is unique to our dataset: it filters words that appear for administrative reasons but have no relation to the details of a patient's diagnosis, like "report", "telephone", "fax", "electronically signed out", etc.

The stop words are contained in `/pathologynotes/utilities/stop_words.py`.

The stop words list for this project: `"a", "an", "the", "and", "for", "that", "of", "outside", "out", "with", "in", "to", "these", "is", "are", "by", "x", "atient", "name", "mrn", "case", "report", "umhc", "comment", "s", "signed", "electronically", "note", "p", "f", "m" "fellow", "clinical", "laboratory", "room", "uhealth", "pathology", "path", "client", "fax", "phone", "telephone", "md", "dr", "phd", "mba", "pathologist", "attending", "i", "ii", "iii", "department", "received", "sw", "nw", "name", "dob", "gender", "medical", "director", "demographics", "ave", "surgical", "physician", "location", "final", "see", "description", "provided", "status", "ordered", "addendum", "patient", "resident", "examination", "issuance", "involved", "history", "diagnosis", "release"`

	No processing	Formatting with re	re + Filtering stop words
Average characters per report	5886.5	4131.1	2959.0
Average words per report	906.27	696.30	472.08

7. Data management.

The dataset, even after preprocessing, still has aspects that are not conducive to machine learning. In particular, it has severe class imbalance, and some minority classes have almost no data available. To address these issues, I've written a few helper functions which can be found in the `/utilities/` directory.

A. Fetch subset.

Because of the breadth of the dataset (i.e., 1394 unique diagnoses, or 141 top-level diagnoses), it doesn't make sense to apply ML methods to the entire dataset, since 1394 or even 141 is too many classes when considering the size of the dataset. Additionally, even if we use only the top-level classifications and set the number of classes at 141, many of those classes have very few examples. D20 has only 2 examples, and C39 has only 4 examples. Of the 141 top-level classifications:

- 12 classes have fewer than 50 examples
- 32 classes have fewer than 100 examples
- 70 classes have fewer than 500 examples
- 91 classes have fewer than 1000 examples

Clearly, there is not sufficient data to train a 141-class classifier. As mentioned earlier, by slicing a different location from the ICD-10 code, we can build datasets with different sets of class labels to train unique classifiers. For examples, we might consider a subset that contains only C34 codes (lung cancer) or C50 codes (breast cancer).

I wrote a custom helper function called `fetch_subset()` to create such subsets. The function takes three parameters:

- "`data`", a pandas dataframe whose second-to-last column contains target label
- "`code`", the IDC-10 code to be included/excluded in the new dataset
- "`mode`" indicates whether to include or exclude entries with the specified ICD-10 code

For example:

```
fetch_subset(data, "C50", mode='include')
```

returns all entries whose ICD-10 codes contain C50, while:

```
fetch_subset(data, "C50.9", mode='exclude')
```

returns all entries that don't contain C50.9.

This function is in `/pathologynotes/utilities/fetch_subset.py` and can be called directly from the command line with the following usage:

```
python utilities/fetch_subset.py <data tsv> <code> [<savefile>] [<include/exclude>]
```

`<data tsv>` is a .tsv file containing a dataset, where the final two columns are the target label and the text data, respectively. `code` is identical to the above.

The final two parameters are optional. By default, the function will save the new dataset as `~/pathologynotes/fetched_subset.py`. If `savefile` is specified, it will save it to `~/pathologynotes/savefile`, where `savefile` could be a path into a directory.

The default for the final parameter is `include`.

The fetch_subset.py function, while useful in its current state, could be improved to accept a list of classes for inclusion or exclusion. As is, if you wanted to create a dataset that contains only examples from C34 and C50, your options would be:

1. Create a dataset for each using 'include' and merge the two datasets.
2. Repeatedly use 'exclude' to remove all classes except C34 and C50.

B. Balance classes.

If classes are imbalanced, a classifier will learn to simply pick the majority class most of the time to minimize loss. Recall the hypothetical classifier that predicts with 99% accuracy whether a patient has a rare disease simply by always giving a negative diagnosis. For these reasons, balancing the number of samples in each class may improve the classifier's macro accuracy, usually at the cost of micro accuracy.

The function `balance_classes(X, y, max_class_size=None)` is contained in `/pathologynotes/utilities/resample.py`

The custom function I wrote to perform the upsampling wraps the `resample()` function from `sklearn.utils`. It accepts either dense `numpy` arrays (such as those output by Word2Vec) or `scipy` CSR sparse arrays (such as those output by TF-IDF).

If `max_class_size=N`:

1. Classes with more than N datapoints are downsampled: a sample of size N is randomly selected from that class.
2. Classes with fewer than N datapoints are upsampled: randomly selected datapoints from each minority class are duplicated until the class is of size N.

If `max_class_size=None` (which is the default):

1. All minority classes are upsampled to the size of the largest class.

On any dataset, three options are worth experimenting with:

1. No resampling.
2. Resampling to a fixed size (e.g. 1000 examples per class)
    a. This may weaken performance due to the loss of data for classes larger than 1000 datapoints.
3. Resampling all classes to the size of the largest class.
    a. Many duplicate examples in the train set could degrade performance.
    b. This approach can drastically increase the number of examples, increasing time and space complexity.

8. Datasets.

A. Abstracts dataset (baseline).

For deep learning methods, I used the PubMed Abstracts dataset as a baseline. This dataset, stored in `/pathologynotes/data/labeled_abstracts_reduced_8000.tsv`, contains 8000 datapoints, each of which is an abstract from a scientific paper accompanied by a label indicating the subject matter. The datapoints fall into 8 classes: chemistry, diagnosis, genetics, metabolism, pathology, physiology, psychology, or surgery.

The text has already been pre-processed. An example datapoint from the physiology class:

> the ciliate loxodes possesses a number of vesicles at its anterior dorsal margin . these so called muller vesicles contain a spherical inclusion muller body in which lie crystals of barium salts . the muller body is connected via a stalk to the wall of its vesicle . it is presumed to function as a stato organelle and to respond by visible motions to changes of the direction of gravity . we have attempted to document the motion of the muller body with respect to the direction of the gravity vector . living cells moving in a horizontal or a vertical plane have been viewed under the light microscope with differential interference contrast documented on video film and sequences of single frames have been evaluated . apparent excursions of the muller body by about floattoken m corresponding to a deviation of 10 at the base of the stalk are observed in cells moving in a horizontal plane . no larger excursions have been seen in the vertical plane . implications of this result for a model of the stato organelle are discussed .

The class is perfectly balanced: 8 classes with 1000 datapoints per class. On a balanced dataset, micro and macro accuracy will be approximately equal.

B. Top-level sites dataset.

Datasets of this type can be generated by slicing the first three characters of each ICD-10 label. For example, all datapoints with labels of the type C50.YYY are united into a single C50 class.

In this dataset, there are 141 top-level classifications, which includes both CXX and DXX codes, which correspond to malignant and benign neoplasms, respectively. The HiSAN paper, however, only uses 68 classes (from C00 to C80). For the sake of comparison, I created a dataset that contains only examples from these classes. This new dataset has 67 classes, because our dataset does not contain C42 (Hematopoietic and reticuloendothelial systems). According to one of the NIH's SEER training modules, "C42 is a vacant category in ICD-10 but is used in ICD-O to designate several topographic sites within the hematopoietic and reticuloendothelial systems. This category serves principally as the topography site for most of the leukemias and related conditions classified to C90-C95 in ICD-10."[7]

To make the comparison more exact, I could have merged all C90-C95 into a single class and kept it in the dataset.

[7] https://training.seer.cancer.gov/coding/differences/other.html

In total, the 67-class dataset that I used has about 92,000 datapoints, about half of the full dataset. It has the following distribution of datapoints per class:

```
C00:  124       C01:  657       C02: 1115       C03:   83       C04:  255       C05:  255
C06:  883       C07:  433       C08:  172       C09:  813       C10:  834       C11:  221
C12:   45       C13:  228       C14:  212       C15: 1141       C16: 1016       C17:  162
C18: 3661       C19:  548       C20:  791       C21:  469       C22: 2110       C23:  137
C24:  362       C25: 3006       C26:   55       C30:  383       C31:  371       C32: 1605
C33:   40       C34: 5720       C37:   67       C38:   91       C40:  228       C41: 1231
C44: 7735       C47:   69       C48:  330       C49: 3470       C50: 16791      C51:  151
C52:   86       C53: 1087       C54: 1446       C55:  456       C56: 1366       C57:  133
C60:   94       C61: 6246       C62:  716       C63:   23       C64: 2956       C65:  237
C66:  256       C67: 3450       C68:  609       C69:  494       C70:   44       C71: 1666
C72:   53       C73: 2888       C74:  145       C75:   51       C76: 2044       C77: 3459
C80: 3941       >>> _
```

There is still significant class imbalance. The smallest class (C63) has only 23 examples, while the largest class (C50) has nearly 17,000.

C. Subsites datasets.

The subclassification data from ICD-10 codes for a particular class can be sliced to create additional datasets. For example, the breast cancer subsites dataset contains examples of the type C50.Y, where the value of Y (0-9) encodes information about the subsite of the tumor within the breast.

The only subsites datasets I used were those for C50 and C34, although others could easily be constructed.

For the C50 subsites dataset, I excluded examples of C50.9 (about 10,000), which correspond to an unspecified site within the breast. On the 8-class dataset of the remaining 6,000 datapoints, I searched the datapoints of each class for key words that could indicate the subsite. Key words list for each class:

- C50.0: "nipple", "areola"
- C50.1: "center", "central"
- C50.2: "upper", "inner"
- C50.3: "lower", "inner"
- C50.4: "upper", "outer"
- C50.5: "lower", "outer"
- C50.6: "axillary", "tail"
- C50.8: "overlapping"

If any of a class's key words were found in a certain example, I considered that example "classifiable". Here are the portions of each class for which examples contained one or more of the keywords:

- C50.0: 0.233
- C50.1: 0.149

- C50.2: 0.170
- C50.3: 0.155
- C50.4: 0.194
- C50.5: 0.141
- C50.6: 0.467
- C50.8: 0.025
- Total percent “classifiable”: 16.29%

This is a rough analysis of the class, and there may be better choices for key words, but this gives some indication that classification on this dataset is a difficult task. In the HiSAN paper, as part of pre-processing, the authors standardize clock-time references, which are used to indicate location within the breast. It’s possible that doing this would improve performance on this task, especially for the classes C50.2-C50.5, which correspond to each of the four quadrants of the breast.

9. Unsupervised learning.

Consider the C50 dataset. The ICD-10 codes for breast cancer are set up so that, in the sub-classification, the first digit encodes the site, the second digit encodes the patient’s gender, and the third digit encodes the laterality (left or right breast). So, I conducted clustering on the C50 dataset to see if the clusters could capture the information contained in the ICD-10 codes.

The C50 labels have a three-digit subclassification. The first digit indicates the site, the second indicates the patient’s gender, and the third indicates laterality. In particular:

- C50.0: Malignant neoplasm of nipple and areola
- C50.1: Malignant neoplasm of central portion of breast
- C50.2: Malignant neoplasm of upper-inner quadrant of breast
- C50.3: Malignant neoplasm of lower-inner quadrant of breast
- C50.4: Malignant neoplasm of upper-outer quadrant of breast
- C50.5: Malignant neoplasm of lower-outer quadrant of breast
- C50.6: Malignant neoplasm of axillary tail of breast
- C50.8: Malignant neoplasm of overlapping sites of breast
- C50.9: Malignant neoplasm of breast of unspecified site

- C50.X1: Female
- C50.X2: Male

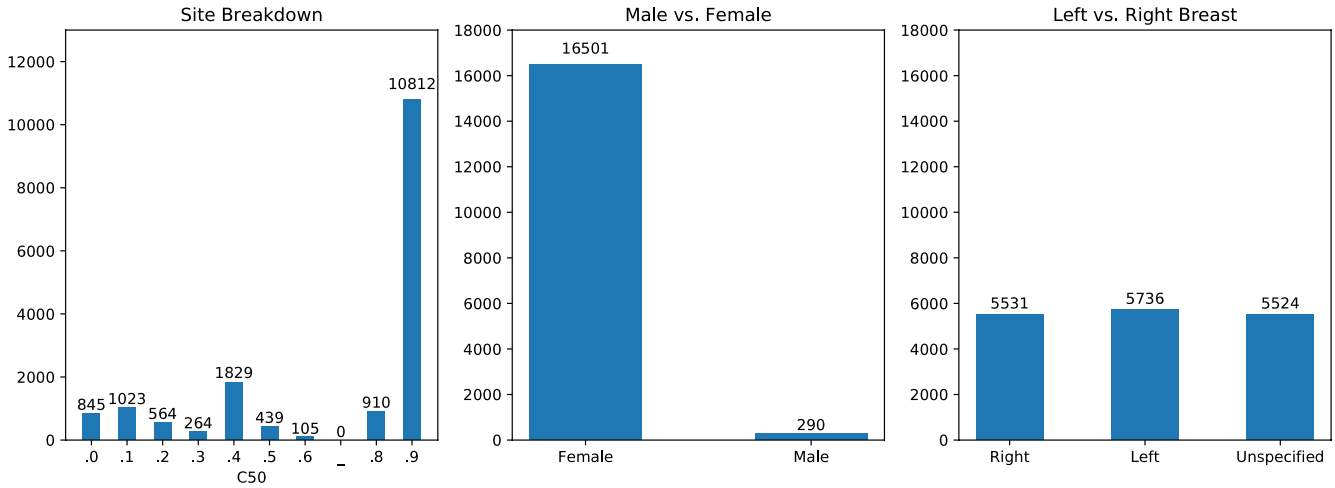
- C50.XX1: Right breast
- C50.XX2: Left breast
- C50.XX9: Unspecified breast

First, I read in the data from icd10notes.txt using pandas, discarded the patient IDs, and fetched the C50 subset, which contains 16,791 datapoints. I ran word frequencies on the dataset. Here were the 10 most common words:

WORD	COUNT
the	332666
and	258648
in	208557
of	207654
for	152945
is	141435
received	137182
floattoken	127489
1	126782
are	125696

"report" appears at 12, "md" appears at 13, "breast" appears at 14, and "carcinoma" appears at 16.

Next, I generated histograms to show the distribution of the labels:

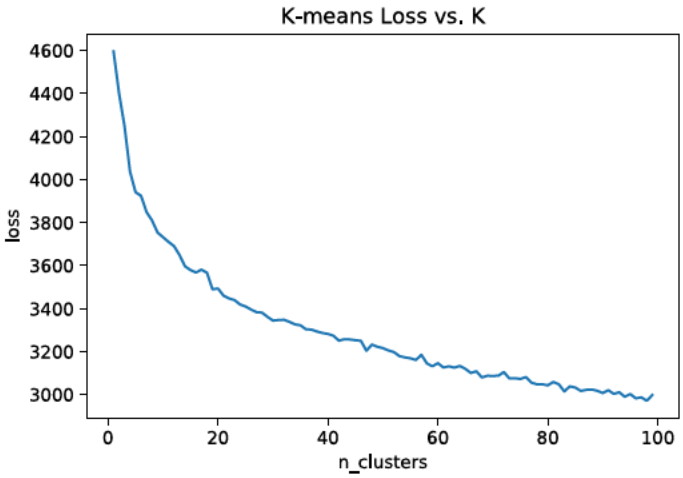


The majority of datapoints are C50.9XX, which indicates an unspecified site. Since the labels for such datapoints don't provide information, they won't be useful. I fetched a new subset of the data that excludes all datapoints with a label that contains C50.9.

The new dataset contains 5979 datapoints. After creating three sets of class labels (one for each of site, gender, laterality), I pre-processed the text by applying the substitutions with re indicated previously.

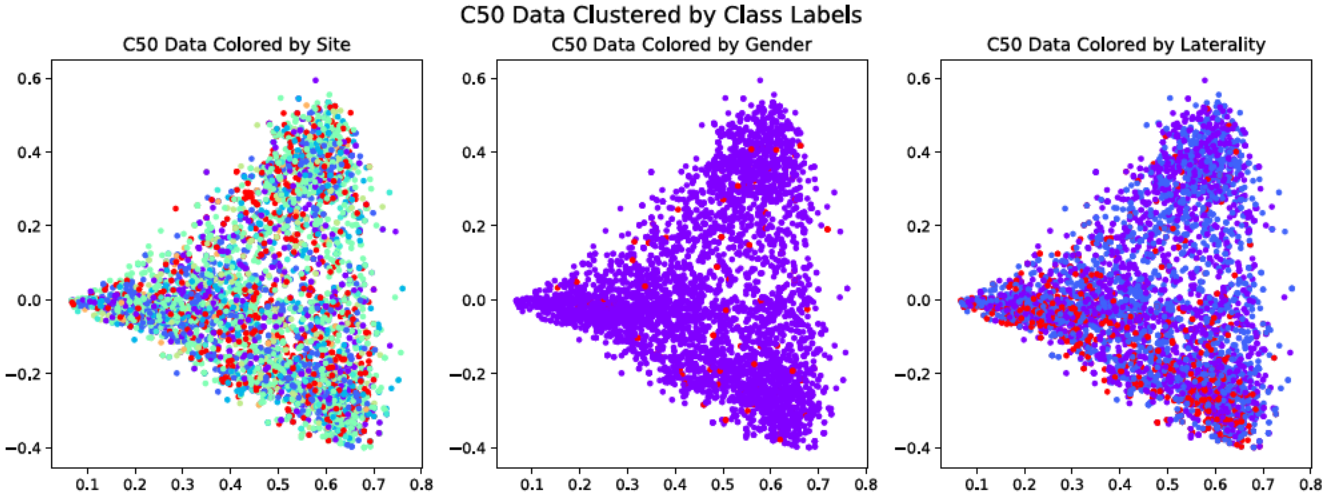
To vectorize the text data, I applied TfidfVectorizer from sklearn, with the custom list of stop words passed as a parameter to the vectorizer. The docs were embedded sparsely in 11381 dimensions. On average, each vector contained 128.49 nonzero elements.

I then applied the clustering algorithm k-means to the vectorized dataset, letting the number of clusters vary from 1-100, and tracked the loss for each execution of the algorithm. The generated loss curve would be used to select an optimal value for K. However, it doesn't flatten out for a reasonable number of clusters, which indicates that either the data is too noisy or does not contain natural clusters.



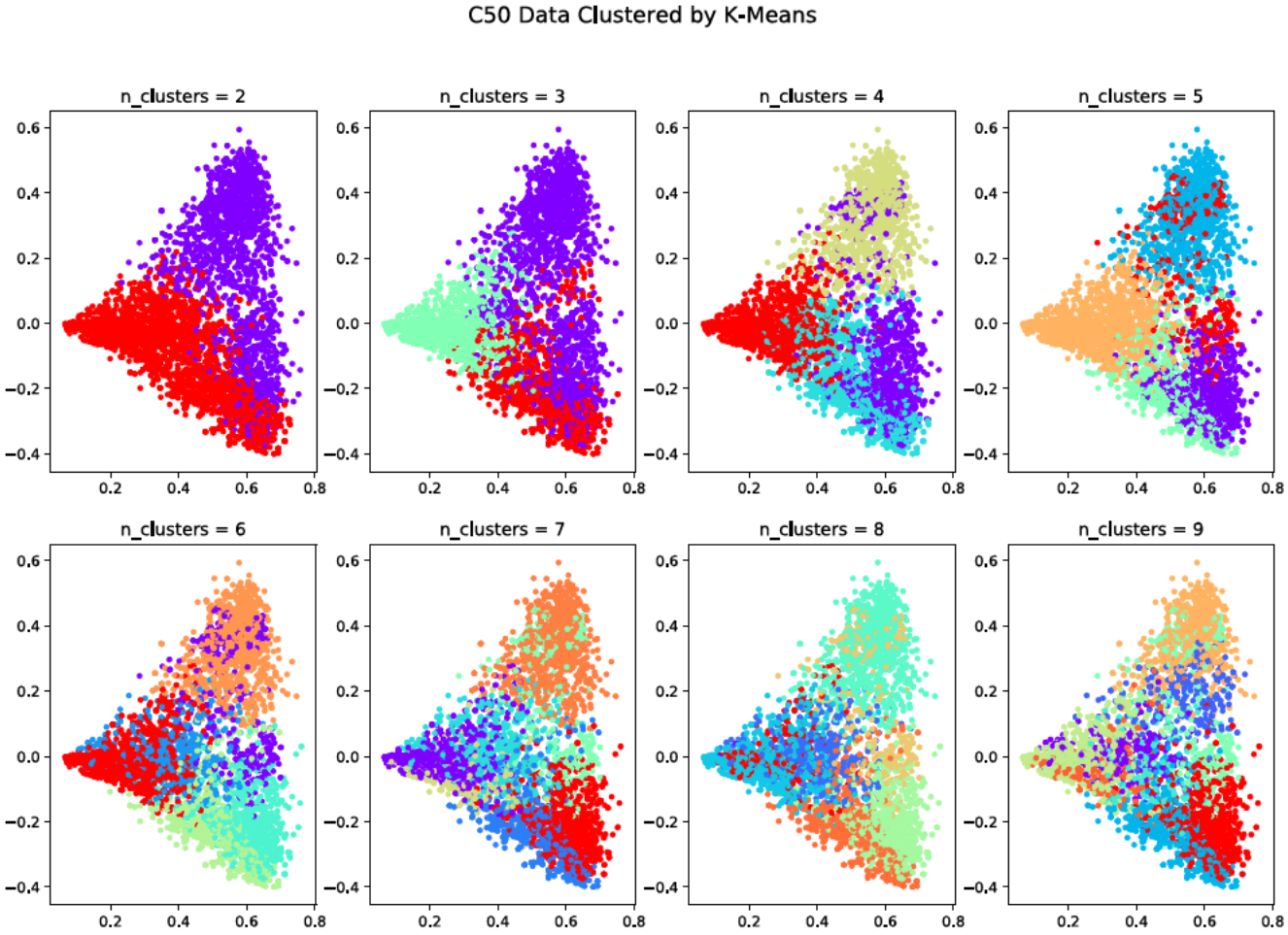
To visualize the clustering, we reduce the data from 11381 dimensions to 2 dimensions so that it can be plotted. This dimensionality reduction was achieved using TruncatedSVD from sklearn, which works for sparse output like that received from TfidfVectorizer.

I plotted the reduced dimensionality dataset and colored it first by the class labels:



However, datapoints from each class appear all over the range of the datapoints. The position of a datapoint in the vector space seems to have nothing to do with the information contained in its ICD-10 label.

For reference, here is the plot of the reduced dimensionality dataset colored according to the output of the k-means algorithm:



Clearly, there is no relationship between the coloring of the two sets of plots. In conclusion, either:

A. The region of the vector space in which a document appears has no relationship with its ICD10 labels; or
B. The data is too noisy, even after preprocessing, to see such a relationship.
C. TfidfVectorizer is too simplistic to effectively extract the information.

Conclusion A does not necessarily mean that vector space machine learning approaches cannot work: the relationships between points in the high-dimensional vector space may hold the data contained in the ICD10 labels. Supervised learning methods may have more success.

I repeated a similar set of experiments on a C34 subset, which contains pathology reports for malignant neoplasms of the bronchus and lung. The dataset contains 5720 datapoints.

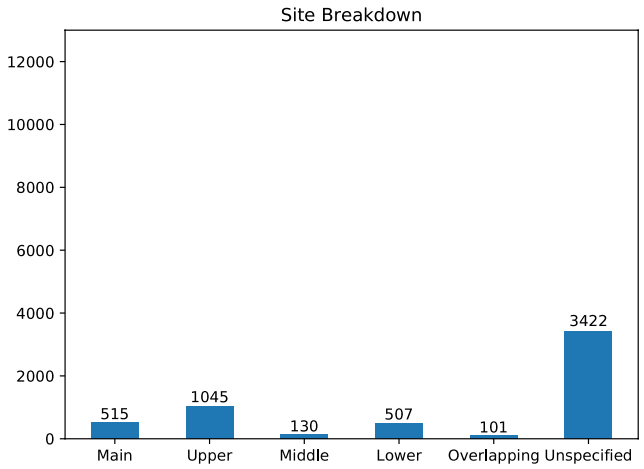
When I calculated the word frequencies, the most common words list looked much the same as that for the C50 dataset. The most common medical word, however, rather than being “breast”, was “lymph”.

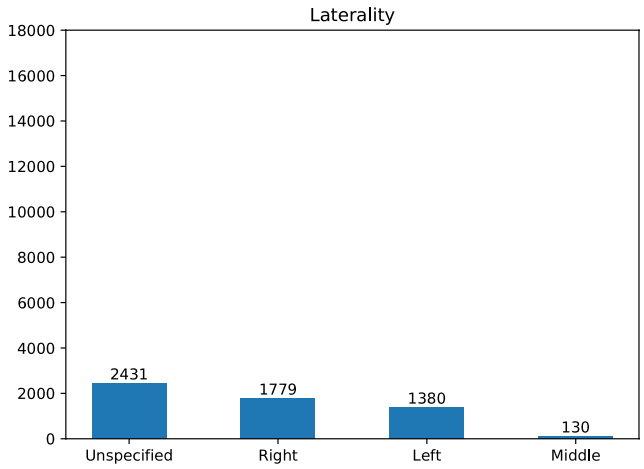
The C34 codes have a two-digit subclassification: the first digit indicates the site, while the second indicates laterality. In particular:

- C34.0: Malignant neoplasm of main bronchus
- C34.1: Malignant neoplasm of upper lobe, bronchus, or lung
- C34.2: Malignant neoplasm of middle lobe, bronchus, or lung
- C34.3: Malignant neoplasm of lower lobe, bronchus, or lung
- C34.8: Malignant neoplasm of overlapping sites of bronchus or lung
- C34.9: Malignant neoplasm of unspecified part of bronchus or lung

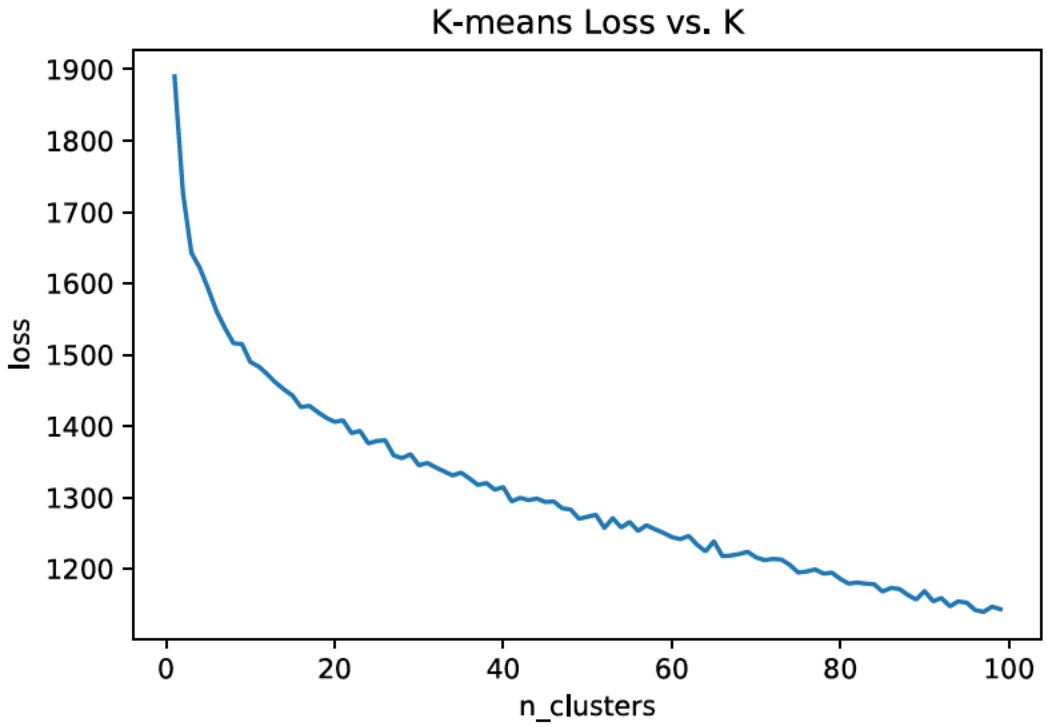
- C34.X0: Unspecified bronchus or lung
- C34.X1: Right bronchus or lung
- C34.X2: Left bronchus or lung

Note: C34.2 does not break down into C34.20, C34.21, and C34.22, and I'm not sure why. Couldn't the tumor be in the middle lobe of the left lung or the middle lobe of the right lung? For the purposes of the following histograms, I’ve classified the laterality of all C34.2 datapoints into their own class, called “Middle”, rather than combining them with the unspecified laterality class. The dataset labels have the following distribution:

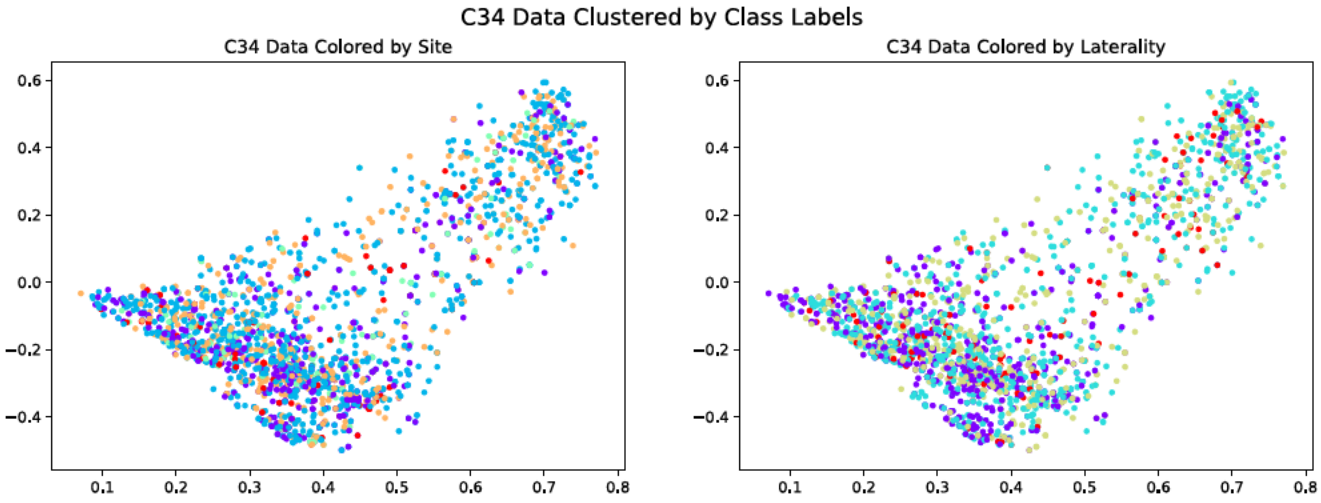




After text processing, TFIDF vectorization (using the same stop words list) embedded the documents sparsely in 8674 dimensions, with an average of 57.54 nonzero elements per datapoint. Running k-means on the vectorized dataset while letting the number of clusters vary from 1-100 generated a loss curve with the same problems as that from the C50 data.

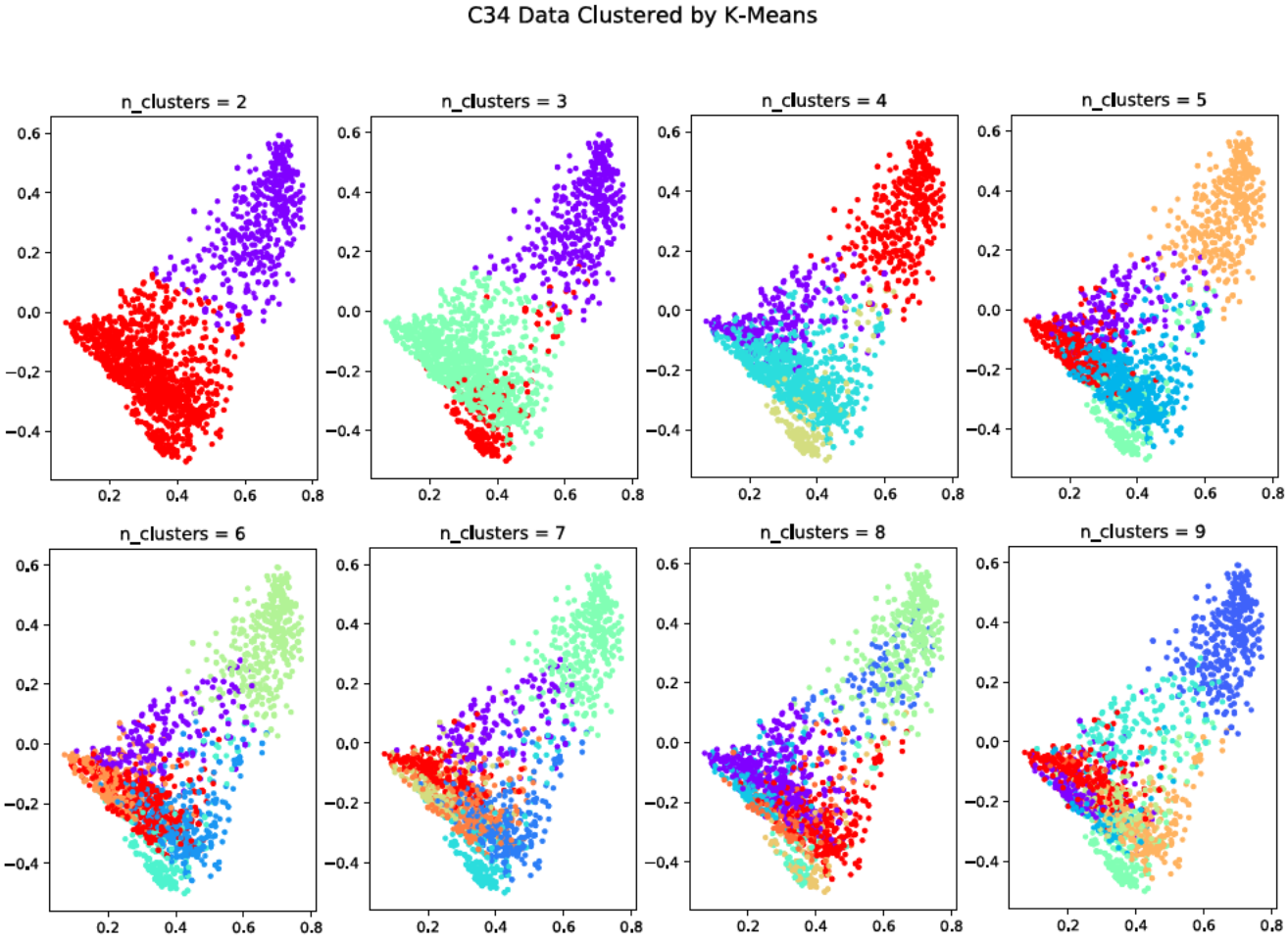


Dimensionality reduction to 2D using TruncatedSVD yields the following plots when colored according to class:



It appears that, once again, there is no relationship between the information contained in the ICD-10 label and the region in the vector space in which the document is embedded.

For reference, here are the plots of the same vectorized dataset, reduced to 2 dimensions, and colored according to clustering with k-means:



This set of experiments gives additional support to the conclusions I drew regarding the C50 dataset. A clustering approach is unlikely to be useful, at least without additional techniques for removing noise from the data.

10. Traditional supervised learning.

In the github repository accompanying their paper, Gao et al. provide the script "traditional_ml.py", which I have adapted for the purposes of this project. The modified script has the following usage:

```
python traditional_ml.py <filepath txt> <label_start>
[label_stop]
```

The label start and stop arguments allow the user to flexibly slice ICD-10 labels from the given dataset to generate classes that correspond to different elements of the label. If `label_stop` is omitted, it takes a value of `label_start + 1` by default.

For example, one could slice the first character in the subclassification of the C50 labels, generating a set of classes corresponding to the subsite within the breast where the tumor appears. Or one could slice the two-digit top level classification from the full dataset to generate a set of classes that indicates the general site where the tumor occurs.

The script first reads in the data, applying text pre-processing and label slicing to generate the lists `docs` and `labels`. Then, it applies TfidfVectorization and filters stop words. If the labels are not integers, it encodes them as such using LabelEncoder from sklearn.

The classifiers are evaluated using ten-split cross fold validation, implemented with the StratifiedKFold object from sklearn. The run_classifier() method takes five arguments:

- `X`, the vectorized documents
- `y`, the target class labels
- `kf`, a StratifiedKFold object
- `clf`, the classifier object
- `balance`, a Boolean that indicates whether to apply class balancing using resampling. By default, it is set to `False`.

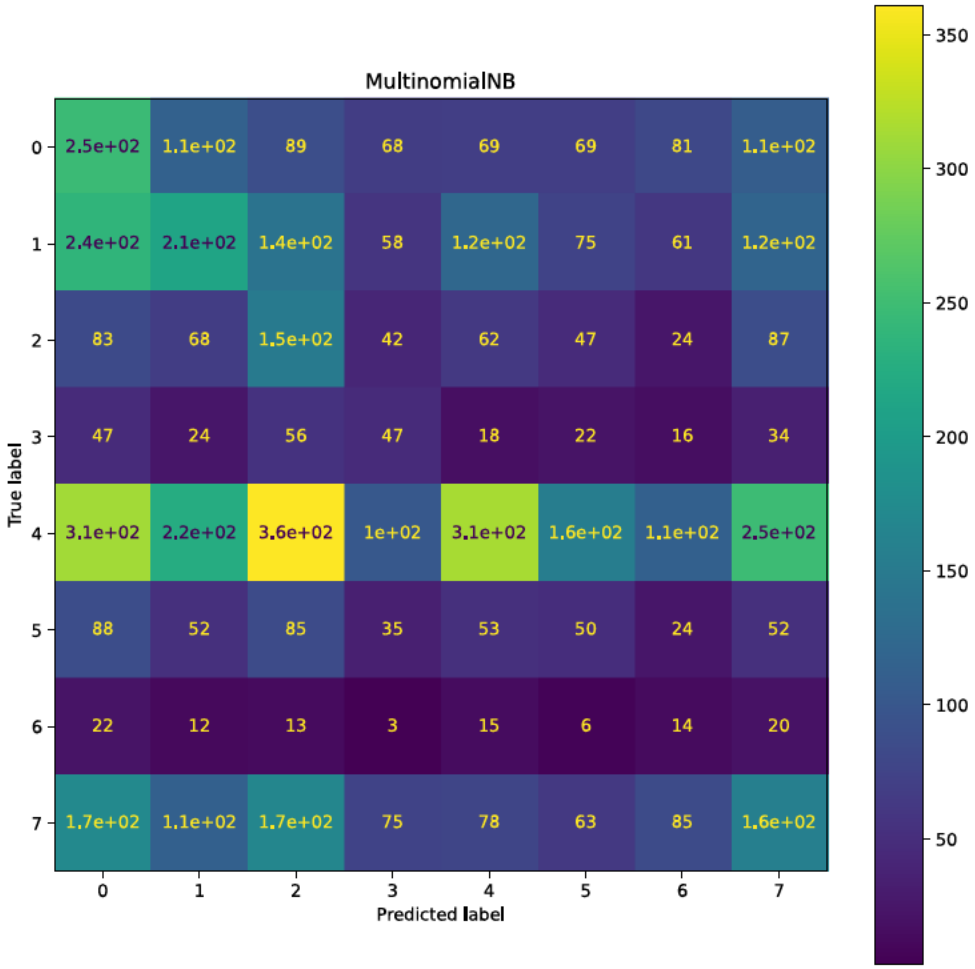
For each of K folds, the function first splits the data into train and testing (validation) sets, balances the training sets with resampling if required, fits to the training set, and then predicts values for the target set. The predicted labels are used to score the classifier (using macro F1 score from sklearn) on this fold and to generate a confusion matrix. Since, in each fold, the classifier is only validated on 1/K of the dataset, the final confusion matrix is the sum of the confusion matrices from each fold. The final score of the classifier, however, is the average of the scores from each of the folds.

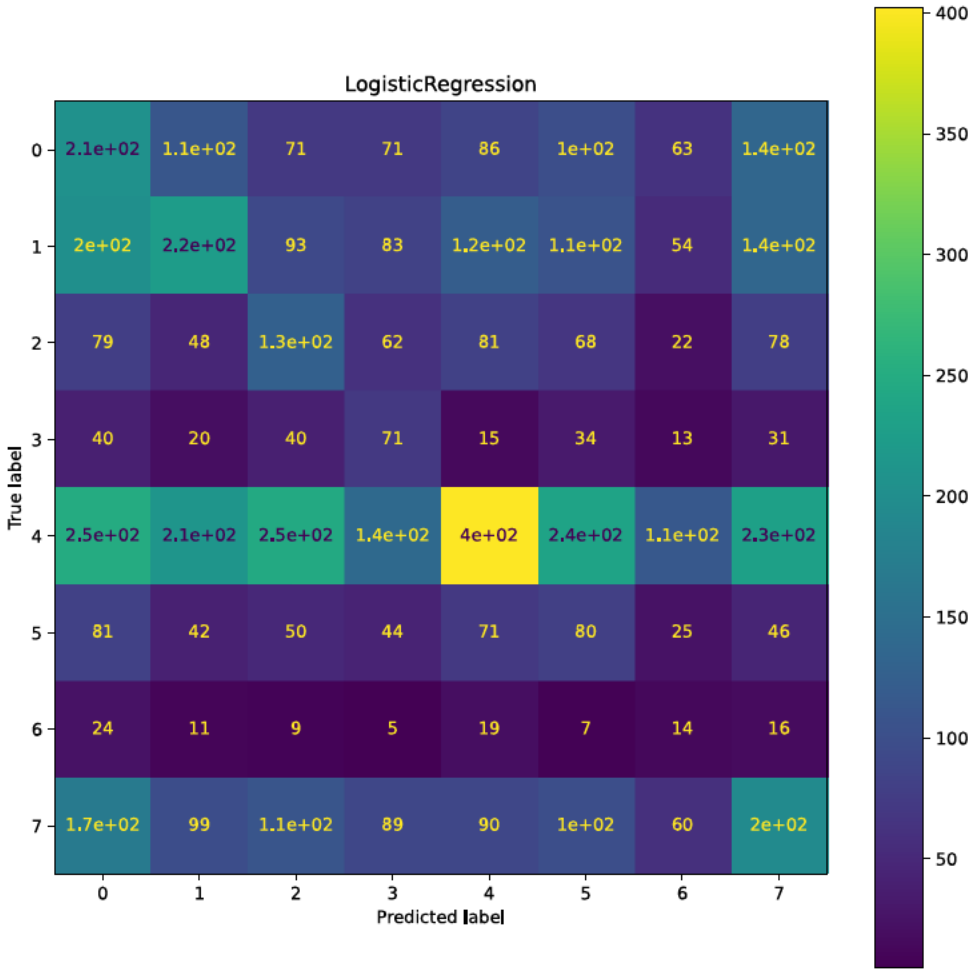
Using this paradigm with 10-fold cross validation, I evaluated four different classifiers' ability to predict the subsite of C50 pathology reports. The dataset has 5797 datapoints distributed (unequally) across 8 classes, because I excluded the reports where the subsite was not specified. Four traditional ML classifiers were imported from sklearn: Multinomial Naïve Bayes, Logistic Regression, Random Forest Classifier, and Support Vector Classifier. MultinomialNB does not have a `class_weight` attribute that can be set to `'balanced'`,

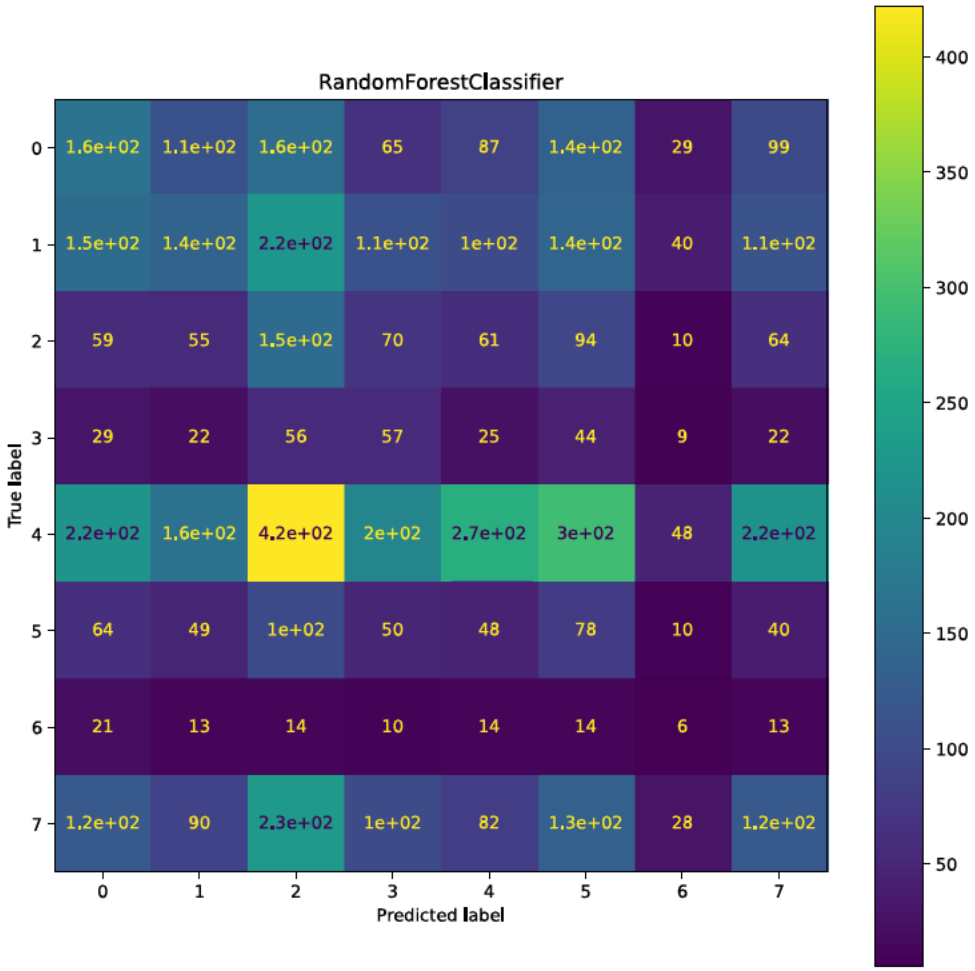
so `balance_classes()` had to be called on each train set before that classifier was fit to the data. The Random Forest Classifier used 100 estimators, each with a max depth of 5. For the latter three, I specified `random_state=0` for the sake of reproducibility.

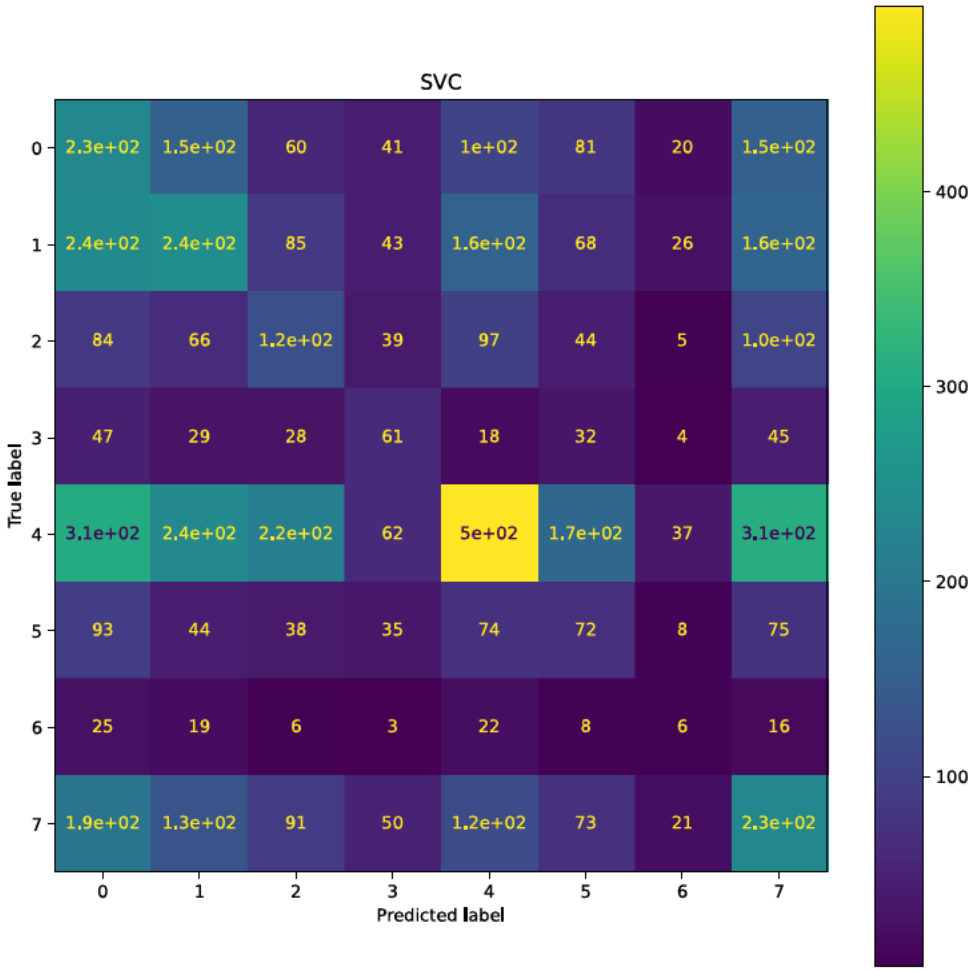
Classifier	Macro F1 Score
Multinomial Naïve Bayes	17.2166
Logistic Regression	19.3076
Random Forest Classifier	14.3541
Support Vector Classifier	20.6868

Here are confusion matrices for each of the classifiers:




LogisticRegression
True label
Predicted label
0 1 2 3 4 5 6 7
2.1e+02 1.1e+02 71 71 86 1e+02 63 1.4e+02
2e+02 2.2e+02 93 83 1.2e+02 1.1e+02 54 1.4e+02
79 48 1.3e+02 62 81 68 22 78
40 20 40 71 15 34 13 31
2.5e+02 2.1e+02 2.5e+02 1.4e+02 4e+02 2.4e+02 1.1e+02 2.3e+02
81 42 50 44 71 80 25 46
24 11 9 5 19 7 14 16
1.7e+02 99 1.1e+02 89 90 1e+02 60 2e+02
400 350 300 250 200 150 100 50


RandomForestClassifier
True label
Predicted label
0
1
2
3
4
5
6
7
1.6e+02 1.1e+02 1.6e+02 65 87 1.4e+02 29 99
1.5e+02 1.4e+02 2.2e+02 1.1e+02 1e+02 1.4e+02 40 1.1e+02
59 55 1.5e+02 70 61 94 10 64
29 22 56 57 25 44 9 22
2.2e+02 1.6e+02 4.2e+02 2e+02 2.7e+02 3e+02 48 2.2e+02
64 49 1e+02 50 48 78 10 40
21 13 14 10 14 14 6 13
1.2e+02 90 2.3e+02 1e+02 82 1.3e+02 28 1.2e+02
400
350
300
250
200
150
100
50



For 6 of the 8 classes, SVC correctly classifies a plurality of the datapoints in that class; that is, it puts more of the datapoints in the correct class than it puts in any other single class. However, for no single class does it approach even 50% accuracy.

It seems that we're once again running into the issue of not having enough data for each class. Class 6 only has 105 datapoints, which isn't enough for the classifier to learn from, a reality that is reflected in the abysmal performance of each classifier on that class.

Ultimately, the results aren't great, but they are better than random chance. The classifiers were able to find at least some association between the text contained in the pathology reports and the location of the neoplasm within the breast as given in the ICD-10 code.

It's likely that the data is still too noisy for these traditional classifiers to perform well. To move forward, we could try additional pre-processing to remove noise from the dataset. However, deep learning methods offer another potential solution, as they can learn how best to extract features from the raw text, rather than requiring extensive human data preprocessing and feature engineering.

11. Deep learning.

Deep neural networks, like those presented in the papers by Gao et al., are a promising solution for dealing with the noise in the data. Included with the papers is tensorflow python code for a text convolutional neural network (CNN), hierarchical attention network (HAN), hierarchical convolutional attention network (HCAN), and hierarchical self-attention network (HiSAN).

A. Basic pipeline.

Input to these architectures is generated using the feature_extraction.py script provided by the papers. I changed the data input pipeline to work for our dataset and added text preprocessing. The script uses Word2Vec from gensim to generate a vocabulary matrix. The script then uses the vocabulary matrix to encode the documents in numerical form, with each word in the document replaced by an integer that refers to that word's row index in the vocabulary matrix. I added stop word filtering to this process. Prior to training, the dataset is divided into training, validation, and testing sets. Then, class balancing can be applied using resampling. Training data is shuffled prior to each epoch.

B. Models.

The text convolutional neural network (CNN) provided in the authors' repository is written in tensorflow V1; however, tensorflow is now in V2. Some backwards compatibility is manageable via the `tf_upgrade_v2` command that automatically upgrades old scripts. However, this command does not fix all issues, and extensive debugging may be required. For an old configuration of the anaconda environment, the CNN script worked. However, upon upgrading to a unified environment, the CNN script was broken beyond hope for repair. I chose to discard it, since its only utility was as a point of comparison for the more sophisticated architectures, which are all but guaranteed to exceed whatever performance it would have had. Even if I spent the time to debug the CNN, upkeep on a script written in tensorflow V1 would be untenable.

The hierarchical attention network (HAN) is also written in tensorflow V1. I again chose not to waste my time extensively debugging an architecture that would likely provide inferior performance.

The hierarchical convolution network (HCAN), while written in tensorflow V1, did work after using the upgrade command and doing some debugging. In addition to changing the data input pipeline, I edited the format of the accuracy printouts to mimic the format of the HiSAN output.

The hierarchical self-attention network (HiSAN) is written in tensorflow V2, so it required few updates beyond adjustments to the data input pipeline.

C. Hyperparameters.

The following hyperparameters warrant consideration when testing. These hyperparameters are not given as arguments to the job scripts. They can be changed by editing variables at the top of the `main` block of the `tf_hcan.py` and `tf_hisan.py` scripts.

1. *Resampling:*

    Resampling can help combat poor model performance due to imbalanced classes. There are generally three options: no resampling; resampling to a fixed class size (e.g., 1000 datapoints); or upsampling up to the maximum class size. Resampling to a fixed class size may weaken performance due to the loss of data for classes larger than the chosen class size, while resampling all classes to the size of the largest class contributes many duplicate examples to the train set could degrade performance and significantly increases time and space complexity for training the model.

    I tested all three of these configurations on the datasets to mixed results. The different settings had only a small impact on accuracy. I found no single best answer – no resampling was best on the top-level sites task, while resampling to the maximum class size appeared optimal for the breast cancer subsites task. However, the improvements in accuracy were small, and could be accounted for by the stochastic nature of training.

2. *Epochs:*

    Each epoch is one run through the entire training set. Training for more epochs requires more time but offers the model more time to fit to the data, which can improve performance up to a point.

    When performing initial testing and debugging, I used only 10 epochs. Eventually, I changed the number of epochs to 100 for the HiSAN. The HiSAN has a patience feature: if there is no improvement in validation macro accuracy for 5 consecutive epochs, training terminates. Thus, while the hyperparameter was set to 100, the HiSAN trained for different amounts of time on different tasks.

3. *Dropout:*

    Deep learning methods are susceptible to overfitting to the peculiarities of the training set at the expense of general prediction ability (i.e., worse performance on unseen data). Regularization techniques penalize model complexity, favoring simpler models that ultimately have more generalized inference abilities.

The HCAN and HiSAN utilize dropout for regularization applied in two places. First, a certain percentage of word and line embeddings are dropped in each level of the hierarchy. Second, the same percentage of outputs from all self- and target-attention mechanisms are dropped.

According to the authors, "Dropping out random words in the normalized similarity matrix generated after the softmax function reduces overfitting by preventing the attention mechanisms from always learning the same relationships between different words, thereby forcing the attention mechanisms to explore new potential word relationships."

Dropout is only applied during training. When performing inference, the model uses its full power.

I experimented with various values of dropout between 0.1 and 0.4. Although the authors used 0.1 across the board, I found that different values sometimes offered slightly better performance on certain tasks.

The following hyperparameters apply only to the HiSAN:

4. *Batch size:*

Batch size is the number of examples fed to the dataset at a time while training. Default is 32 and I didn't change it.

5. *Attention size:*

Attention size is the embedding dimension of the self-attention mechanisms. Default is 400 and I didn't change it.

6. *Number of attention heads:*

The HiSAN leverages multihead attention, using parallel attention mechanisms, each of which attends to a different portion of the embedding dimension.

While the authors only ever use 8 attention heads, I performed some tests using 8 attention heads and others using 16 attention heads. I didn't see much of a difference except in runtime, with the latter being twice as fast.

D. Results.

On the abstracts dataset, I ran two tests: one each with the HCAN and HiSAN. I used the following hyperparameters:

	HCAN	HiSAN
Epochs	10	25
Dropout	0.2	0.2

Attention Heads	N/A	16
Resampling?	No	No

Resampling is, of course, unnecessary for this dataset because it is perfectly balanced. It has exactly 1000 datapoints for each of its 8 classes.

Time complexity details on this task:

	HCAN (dropout=0.2)	**HiSAN (dropout=0.2, attention_heads=16)**
Epochs	10	25
CPU Time	2492s (41.5 min)	3290s (54.8min)
Avg. CPU Time per Epoch	249.2s (4.15 min)	131.6s (2.2 min)

The HiSAN is approximately twice as fast as the HCAN. The HiSAN also takes advantage of parallelization: its run time is significantly less than its CPU time. The HCAN, in its current state, does not (it has basically the same run time and CPU time), but I suspect that may have been caused by the updates that were made to the script. It could be a quirk of workarounds that were required to make it operational in the TRITON environment.

Results on the abstracts task with no resampling:

	HCAN (dropout=0.2)	**HiSAN (dropout=0.2, attention_heads=16)**
Epochs	10	25
Best Train Micro Acc	0.9263	0.9042
Best Train Macro Acc	0.9265	0.9042
Best Val Micro Acc	0.7106	0.7581
Best Val Macro Acc		0.7595

As expected from a balanced dataset, the micro and macro accuracy are virtually identical. When I ran the test, the HCAN was not configured to output macro accuracy on the dataset. We can infer, however, that the value would be approximately 0.71. The HCAN attains higher training accuracy (despite fewer epochs), but lower validation accuracy. It's possible that, with more epochs (and

perhaps more dropout), the HCAN's validation accuracy could approach that of the HiSAN. However, because these values are intended as baselines to compare with these architectures' performance on other datasets, I didn't spend time optimizing their performance on this dataset. Since there is a sizable gap between training and validation accuracies for both architectures, it's possible that both could benefit from a higher dropout value.

On the top-level site prediction task, whose 67-class dataset has 92,000 datapoints, I ran a number of tests.

Time complexity data for a few of the tests:

	HCAN	HiSAN (no resampling, Attention_heads=8)	HiSAN (no resampling, Attention_heads=16)
Epochs	10	16	21
CPU Time	~55000s (15 hr)	~40000s (11 hr)	~26000s (7hr)
Avg. CPU Time per Epoch	~5500s (1.5 hr)	~2500s (42 mins)	~1250s (21 mins)

Here, the HiSAN is much faster than the HCAN, and the HiSAN with 16 attention heads is almost exactly twice as fast as the HiSAN with 8 attention heads. The attention heads divide up the space of the embedding dimension and run in parallel, so doubling the number of attention heads should about double the speed of the attention computations.

With no resampling, the HCAN attains the following performance on this task:

HCAN	**Dropout=0.1**	**Dropout=0.2**
Train micro acc	0.7488	0.7197
Train macro acc	0.5410	0.4555
Val macro acc	0.3184	0.3352

With 8 attention heads, the HiSAN attains the following validation accuracy (micro; macro) on this task when varying the level of dropout and the resampling technique:

HiSAN with 8 attention heads	No Resampling	Resample to 1k	Upsample to Max (13299 examples)
Dropout=0.1	0.6256; 0.3789		
Dropout=0.15	0.6257; 0.3781		
Dropout=0.2	0.6256; 0.3833		
Dropout=0.3	0.6233; 0.3676		
Dropout=0.4	0.6194; 0.3592	0.5390; 0.3468	0.5513; 0.3517

I first tested upsampling to the max (13,299 datapoints per class) with dropout=0.4 (shown in the bottom right). I used this high value of dropout because there was a serious gap in training and validation accuracy. However, this gap is due to the presence of a large number of duplicates (from upsampling) in the training set, which can be easily fit by the model.

However, when I tested no resampling at the same level of dropout, I found better performance. Consequently, I chose to optimize the value of dropout without resampling. The gains in accuracy from this optimization, however, were small. The authors attained macro accuracy in the low 60's on this task, significantly higher than what I was able to attain. I suspect two factors: one, my text pre-processing is not on par with theirs; two, that our dataset is simply less structured and more difficult to classify.

With 16 attention heads, no resampling, and dropout=0.2 on this task, the HiSAN trained for 21 epochs, and attained training accuracy 0.6475, 0.4555 (micro/macro) and validation accuracy 0.6302, 0.3837 (micro/macro). The 0.3837 achieved here was the best performance on this task from any configuration I used.

Finally, on the breast cancer subsite task, I ran tests with the following hyperparameters:

	HCAN	HiSAN1	HiSAN2	HiSAN3
Epochs	10	45	20	29
Dropout	0.2	0.2	0.2	0.4
Attention Heads	N/A	16	16	16
Resampling?	Max	No	Max	Max

The HCAN, with upsampling to the maximum class size (1448 datapoints per class), dropout=0.2, when trained for 10 epochs, attained training accuracy 0.8335, 0.8313 (micro/macro) and validation accuracy 0.2415, 0.1882 (micro/macro)

The HiSAN (with 16 attention heads), while varying the level of dropout and the resampling technique, attained the following accuracies (micro; macro):

HiSAN	**No resampling**	**Upsampling to Max (1448 datapoints)**
Dropout=0.2	Train: 0.4851; 0.3596 Val: 0.2926; 0.1580	Train: 0.6416; 0.6353 Val: 0.2570; 0.1927
Dropout=0.4	~	Train: 0.5868; 0.5787 Val: 0.2399; 0.1939

Ultimately, 0.1939 was the highest validation macro accuracy attained on this task. On this task, upsampling offers a clear benefit, presumably due to the small size of the dataset.

12. Conclusion and Next Steps

Over the course of this semester, I finished debugging both my utilities and the HCAN and HiSAN scripts; configured a proper anaconda environment; organized the bitbucket repository for other potential users; and tested deep learning approaches on several datasets, evaluating the effects of various hyperparameter tweaks.

There are several paths forward for the project:

1. Evaluate alternative deep learning approaches for the dataset. Pre-trained methods such as ClinicalBERT look especially promising.

2. Continue improving text pre-processing.

3. Get an expert to label some data, then apply semi-supervised learning methods, which operate on datasets with a little bit of labeled data and a lot of unlabeled data. In this way, we can extract other data from the reports: tumor size, stage/grade, etc.

4. Build a binary classifier that can decide whether a pathology report is cancer related. Such a classifier could be a helpful sanity check for researchers working with medical notes datasets.

13. References

Gao, Shang, Arvind Ramanathan, and Georgia Tourassi. 2018. "Hierarchical Convolutional Attention Networks for Text Classification." https://aclanthology.org/W18-3002.pdf.

Gao, Shang, John X. Qiu, Alawad Mohammed, Jacob D. Hinkle, Noah Schaefferkoetter, Hong-Jun Yoon, Blair Christian, et al. 2019. "Classifying cancer pathology reports with hierarchical self-attention networks." Artificial Intelligence in Medicine.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention is All You Need." https://arxiv.org/abs/1706.03762.