Introduction to Clinical Natural Language Processing: Predicting Hospital Readmission with Discharge Summaries

Andrew Long Follow
Jun 4, 2018 · 17 min read ★

wealth of knowledge and insight that can be utilized for predictive models using Natural Language Processing (NLP) to improve patient care and hospital workflow. As an example, I will show you how to predict hospital readmission with discharge summaries.

This article is intended for people interested in healthcare data science. After completing this tutorial, you will learn

- How to prepare data for a machine learning project
- How to preprocess the unstructured notes using a bag-of-words approach
- How to build a simple predictive model
- How to assess the quality of your model
- How to decide the next step for improving the model

I recently read this great paper "Scalable and accurate deep learning for electronic health records" by Rajkomar et al. (paper at https://arxiv.org/abs/1801.07860). The authors built many state-of-the-art deep learning models with hospital data to predict in-hospital mortality (AUC = 0.93–0.94), 30-day unplanned readmission (AUC = 0.75–76), prolonged length of stay (AUC = 0.85–0.86) and discharge diagnoses (AUC = 0.90). AUC is a data science performance metric (more about this below) where closer to 1 is better. It is clear that predicting readmission is the hardest task since it has a lower AUC. I was curious how good of a model we can get if use the discharge free-text summaries with a simple predictive model.

If you would like to follow along with the Python code in a Jupyter Notebook, feel free to download the code from my github.

Model Definition

This blog post will outline how to build a classification model to predict which patients

Massachusetts from 2001 to 2012. In order to get access to the data for this project, you will need to request access at this link (https://mimic.physionet.org/gettingstarted/access/).

In this project, we will make use of the following MIMIC III tables

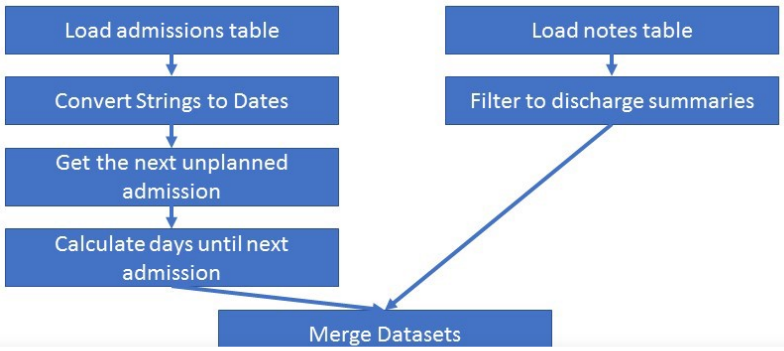
- ADMISSIONS — a table containing admission and discharge dates (has a unique identifier HADM_ID for each admission)
- NOTEEVENTS — contains all notes for each hospitalization (links with HADM_ID)

To maintain anonymity, all dates have been shifted far into the future for each patient, but the time between two consecutive events for a patient is maintained in the database. This is important as it maintains the time between two hospitalizations for a specific patient.

Since this is a restricted dataset, I am not able to publicly share raw patient data. As a result, I will only show you artificial single patient data or aggregated descriptions.

Step 1: Prepare data for a machine learning project

We will follow the steps below to prepare the data from the ADMISSIONS and NOTEEVENTS MIMIC tables for our machine learning project.



```
# set up notebook
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# read the admissions table
df_adm = pd.read_csv('ADMISSIONS.csv')
```

The main columns of interest in this table are :

- SUBJECT_ID: unique identifier for each subject
- HADM_ID: unique identifier for each hospitalization
- ADMITTIME: admission date with format YYYY-MM-DD hh:mm:ss
- DISCHTIME: discharge date with same format
- DEATHTIME: death time (if it exists) with same format
- ADMISSION_TYPE: includes ELECTIVE, EMERGENCY, NEWBORN, URGENT

The next step is to convert the dates from their string format into a datetime. We use the `errors = 'coerce'` flag to allow for missing dates.

```
# convert to dates
df_adm.ADMITTIME = pd.to_datetime(df_adm.ADMITTIME, format = '%Y-%m-
%d %H:%M:%S', errors = 'coerce')
df_adm.DISCHTIME = pd.to_datetime(df_adm.DISCHTIME, format = '%Y-%m-
%d %H:%M:%S', errors = 'coerce')
df_adm.DEATHTIME = pd.to_datetime(df_adm.DEATHTIME, format = '%Y-%m-
%d %H:%M:%S', errors = 'coerce')
```

The next step is to get the next unplanned admission date if it exists. This will follow a

The dataframe could look like this now for a single patient:

	SUBJECT_ID	ADMISSION_TYPE	ADMITTIME
0	A1	EMERGENCY	2100-05-01 10:00:00
1	A1	EMERGENCY	2100-05-20 12:18:00
2	A1	ELECTIVE	2100-07-21 13:11:00
3	A1	EMERGENCY	2100-07-31 09:45:00

We can use the groupby shift operator to get the next admission (if it exists) for each SUBJECT_ID

```
# add the next admission date and type for each subject using groupby
# you have to use groupby otherwise the dates will be from different
subjects
df_adm['NEXT_ADMITTIME'] =
df_adm.groupby('SUBJECT_ID').ADMITTIME.shift(-1)

# get the next admission type
df_adm['NEXT_ADMISSION_TYPE'] =
df_adm.groupby('SUBJECT_ID').ADMISSION_TYPE.shift(-1)
```

	SUBJECT_ID	ADMISSION_TYPE	ADMITTIME	NEXT_ADMITTIME	NEXT_ADMISSION_TYPE
0	A1	EMERGENCY	2100-05-01 10:00:00	2100-05-20 12:18:00	EMERGENCY
1	A1	EMERGENCY	2100-05-20 12:18:00	2100-07-21 13:11:00	ELECTIVE
2	A1	ELECTIVE	2100-07-21 13:11:00	2100-07-31 09:45:00	EMERGENCY
3	A1	EMERGENCY	2100-07-31 09:45:00	NaT	NaN

Note that the last admission doesn't have a next admission.

But, we want to predict UNPLANNED re-admissions, so we should filter out the ELECTIVE next admissions.

1	A1	EMERGENCY	2100-05-20 12:18:00	NaT	NaN
2	A1	ELECTIVE	2100-07-21 13:11:00	2100-07-31 09:45:00	EMERGENCY
3	A1	EMERGENCY	2100-07-31 09:45:00	NaT	NaN

And then backfill the values that we removed

```
# sort by subject_ID and admission date
# it is safer to sort right before the fill in case something changed
the order above
df_adm = df_adm.sort_values(['SUBJECT_ID','ADMITTIME'])

# back fill (this will take a little while)
df_adm[['NEXT_ADMITTIME','NEXT_ADMISSION_TYPE']] =
df_adm.groupby(['SUBJECT_ID'])
[['NEXT_ADMITTIME','NEXT_ADMISSION_TYPE']].fillna(method = 'bfill')
```

	SUBJECT_ID	ADMISSION_TYPE	ADMITTIME	NEXT_ADMITTIME	NEXT_ADMISSION_TYPE
0	A1	EMERGENCY	2100-05-01 10:00:00	2100-05-20 12:18:00	EMERGENCY
1	A1	EMERGENCY	2100-05-20 12:18:00	2100-07-31 09:45:00	EMERGENCY
2	A1	ELECTIVE	2100-07-21 13:11:00	2100-07-31 09:45:00	EMERGENCY
3	A1	EMERGENCY	2100-07-31 09:45:00	NaT	NaN

We can then calculate the days until the next admission

```
df_adm['DAYS_NEXT_ADMIT']=  (df_adm.NEXT_ADMITTIME -
df_adm.DISCHTIME).dt.total_seconds()/(24*60*60)
```

In our dataset with 58976 hospitalizations, there are 11399 re-admissions. For those with a re-admission, we can plot the histogram of days between admissions.

Days between admissions

Now we are ready to work with the NOTEEVENTS.csv

```
df_notes = pd.read_csv("NOTEEVENTS.csv")
```

The main columns of interest are:

- SUBJECT_ID
- HADM_ID
- CATEGORY: includes 'Discharge summary', 'Echo', 'ECG', 'Nursing', 'Physician ', 'Rehab Services', 'Case Management ', 'Respiratory ', 'Nutrition', 'General', 'Social Work', 'Pharmacy', 'Consult', 'Radiology', 'Nursing/other'
- TEXT: our clinical notes column

Since I can't show individual notes, I will just describe them here. The dataset has 2,083,180 rows, indicating that there are multiple notes per hospitalization. In the notes, the dates and PHI (name, doctor, location) have been converted for confidentiality. There are also special characters such as \n (new line), numbers and punctuation.

Since there are multiple notes per hospitalization, we need to make a choice on what notes to use. For simplicity, let's use the discharge summary, but we could use all the notes by concatenating them if we wanted.

```
# filter to discharge summary
```

```
<ipython-input-36-753515103982> in <module>()
----> 1 assert df_notes_dis_sum.duplicated(['HADM_ID']).sum() == 0, 'Multiple discharge summaries per admission'

AssertionError: Multiple discharge summaries per admission
```

At this point, you might want to investigate why there are multiple summaries, but for simplicity let's just use the last one

```
df_notes_dis_sum_last =
(df_notes_dis_sum.groupby(['SUBJECT_ID','HADM_ID']).nth(-1)).reset_in
dex()
assert df_notes_dis_sum_last.duplicated(['HADM_ID']).sum() == 0,
'Multiple discharge summaries per admission'
```

Now we are ready to merge the admissions and notes tables. I use a left merge to account for when notes are missing. There are a lot of cases where you get multiple rows after a merge(although we dealt with it above), so I like to add assert statements after a merge

```
df_adm_notes =
pd.merge(df_adm[['SUBJECT_ID','HADM_ID','ADMITTIME','DISCHTIME','DAYS
_NEXT_ADMIT','NEXT_ADMITTIME','ADMISSION_TYPE','DEATHTIME']],

df_notes_dis_sum_last[['SUBJECT_ID','HADM_ID','TEXT']],
                      on = ['SUBJECT_ID','HADM_ID'],
                      how = 'left')
assert len(df_adm) == len(df_adm_notes), 'Number of rows increased'
```

10.6 % of the admissions are missing (`df_adm_notes.TEXT.isnull().sum() / len(df_adm_notes)`), so I investigated a bit further with

```
df_adm_notes.groupby('ADMISSION_TYPE').apply(lambda g:
g.TEXT.isnull().sum())/df_adm_notes.groupby('ADMISSION_TYPE').size()
```

```
df_adm_notes_clean['OUTPUT_LABEL'] =
(df_adm_notes_clean.DAYS_NEXT_ADMIT < 30).astype('int')
```

A quick count of positive and negative results in 3004 positive samples, 48109 negative samples. This indicates that we have an imbalanced dataset, which is a common occurrence in healthcare data science.

The last step to prepare our data is to split the data into training, validation and test sets. For reproducible results, I have made the random_state always 42.

```
# shuffle the samples
df_adm_notes_clean = df_adm_notes_clean.sample(n =
len(df_adm_notes_clean), random_state = 42)
df_adm_notes_clean = df_adm_notes_clean.reset_index(drop = True)

# Save 30% of the data as validation and test data
df_valid_test=df_adm_notes_clean.sample(frac=0.30,random_state=42)

df_test = df_valid_test.sample(frac = 0.5, random_state = 42)
df_valid = df_valid_test.drop(df_test.index)

# use the rest of the data as training data
df_train_all=df_adm_notes_clean.drop(df_valid_test.index)
```

Since the prevalence is so low, we want to prevent the model from always predicting negative (not re-admitted). To do this, we have a few options to balance the training data

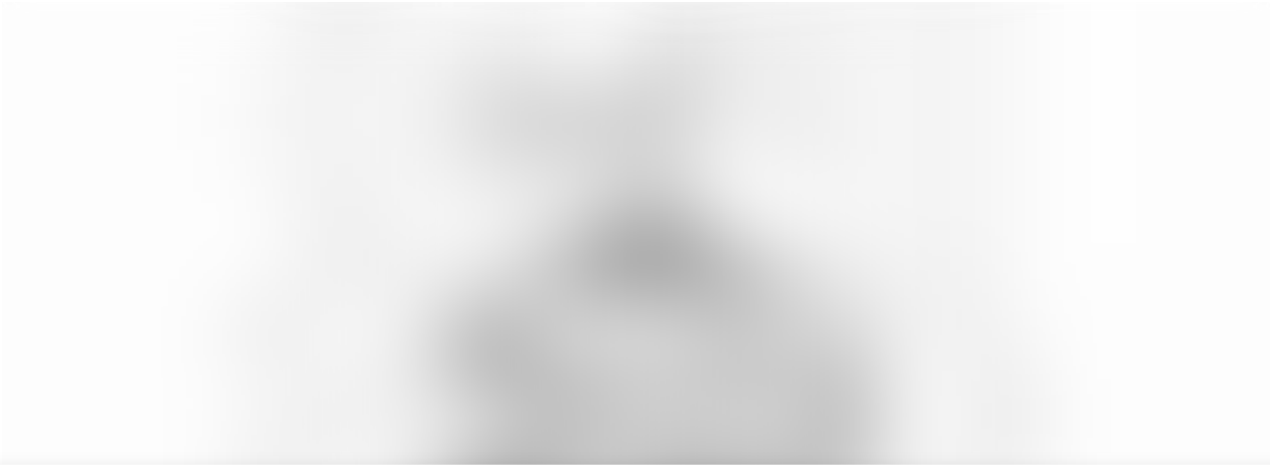
- sub-sampling the negatives
- over-sampling the positives
- create synthetic data (e.g. SMOTE)

```
df_train_pos = df_train_all.loc[rows_pos]
df_train_neg = df_train_all.loc[~rows_pos]

# merge the balanced data
df_train = pd.concat([df_train_pos, df_train_neg.sample(n =
len(df_train_pos), random_state = 42)],axis = 0)

# shuffle the order of training samples
df_train = df_train.sample(n = len(df_train), random_state =
42).reset_index(drop = True)
```

Step 2: Preprocess the unstructured notes using a bag-of-words approach

Now that we have created data sets that have a label and the notes, we need to preprocess our text data to convert it to something useful (i.e. numbers) for the machine learning model. We are going to use the Bag-of-Words (BOW) approach.

BOW basically breaks up the note into the individual words and counts how many times each word occurs. Your numerical data then becomes counts for some set of words as shown below. BOW is the simplest way to do NLP classification. In most blog posts I have read, fancier techniques have a hard time beating BOW for NLP classification tasks.



- how to count the words
- which words to use

There is no optimal choice for all NLP projects, so I recommend trying out a few options when building your own models.

You can do the preprocessing in two ways

- modify the original dataframe TEXT column
- preprocess as part of your pipeline so you don't edit the original data

I will show you how to do both of these, but I prefer the second one since it took a lot of work to get to this point.

Let's define a function that will modify the original dataframe by filling missing notes with space and removing newline and carriage returns

```
def preprocess_text(df):
    # This function preprocesses the text by filling not a number and
replacing new lines ('\n') and carriage returns ('\r')
    df.TEXT = df.TEXT.fillna(' ')
    df.TEXT = df.TEXT.str.replace('\n',' ')
    df.TEXT = df.TEXT.str.replace('\r',' ')
    return df

# preprocess the text to deal with known issues
df_train = preprocess_text(df_train)
df_valid = preprocess_text(df_valid)
df_test = preprocess_text(df_test)
```

The other option is to preprocess as part of the pipeline. This process consists of using a tokenizer and a vectorizer. The tokenizer breaks a single note into a list of words and a

```
word_tokenize('This should be tokenized. 02/02/2018 sentence has
stars**')
```

With output:

```
['This', 'should', 'be', 'tokenized', '.', '02/02/2018', 'sentence',
 'has', 'stars**']
```

The default shows that some punctuation is separated and that numbers stay in the sentence. We will write our own tokenizer function to

- replace punctuation with spaces
- replace numbers with spaces
- lower case all words

```
import string
def tokenizer_better(text):
    # tokenize the text by replacing punctuation and numbers with
spaces and lowercase all words

    punc_list = string.punctuation+'0123456789'
    t = str.maketrans(dict.fromkeys(punc_list, " "))
    text = text.lower().translate(t)
    tokens = word_tokenize(text)
    return tokens
```

With this tokenizer we get from our original sentence

```
['this', 'should', 'be', 'tokenized', 'sentence', 'has', 'stars']
```

across all notes, but for this project let's use the simpler one (I got similar results with the second one too).

As an example, let's say we have 3 notes

```
sample_text = ['Data science is about the data', 'The science is
amazing', 'Predictive modeling is part of data science']
```

Essentially, you fit the CountVectorizer to learn the words in your data and the transform your data to create counts for each word.

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer(tokenizer = tokenizer_better)
vect.fit(sample_text)

# matrix is stored as a sparse matrix (since you have a lot of zeros)
X = vect.transform(sample_text)
```

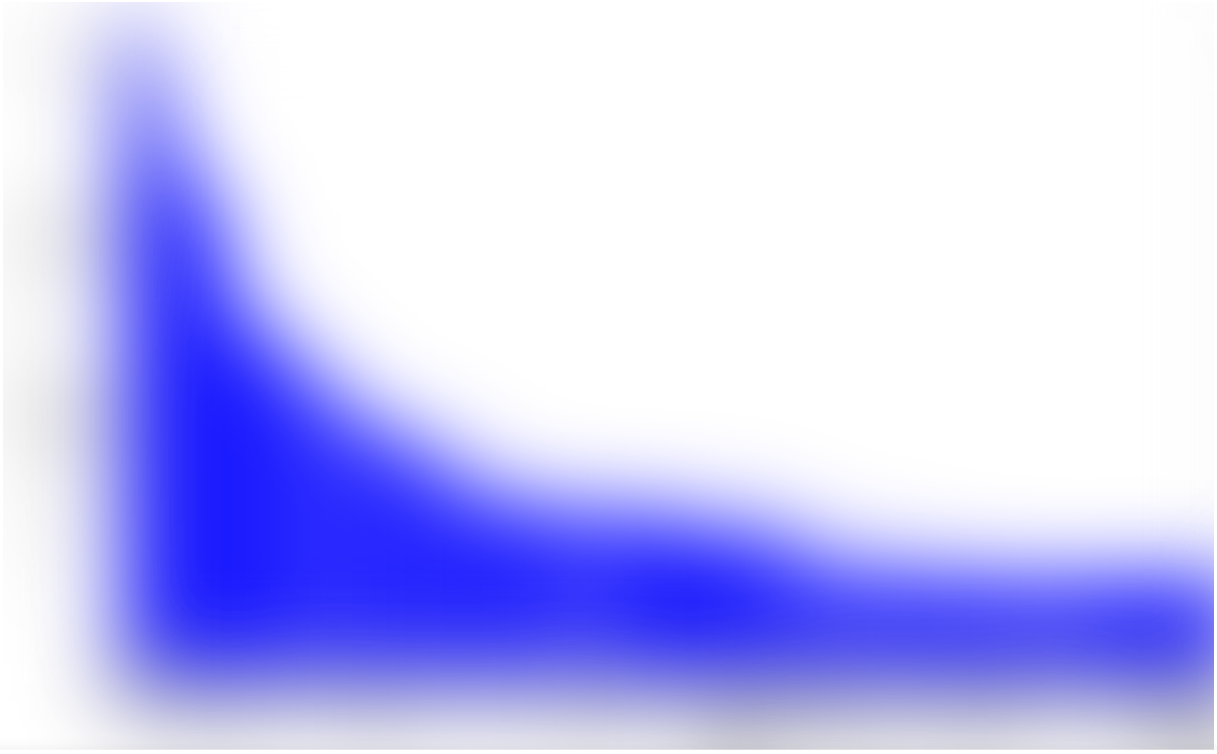
The matrix X will be a sparse matrix, but if you convert it to an array (`X.toarray()`), you will see this

```
array([[1, 0, 2, 1, 0, 0, 0, 0, 1, 1],
       [0, 1, 0, 1, 0, 0, 0, 0, 1, 1],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 0]], dtype=int64)
```

Where there are 3 rows (since we have 3 notes) and counts of each word. You can see the column names with `vect.get_feature_names()`

```
# fit our vectorizer. This will take a while depending on your
computer.

from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer(max_features = 3000, tokenizer =
tokenizer_better)

# this could take a while
vect.fit(df_train.TEXT.values)
```

We can look at the most frequently used words and we will see that many of these words might not add any value for our model. These words are called stop words, and we can remove them easily (if we want) with the CountVectorizer. There are lists of common stop words for different NLP corpus, but we will just make up our own based on the image below.



Feel free to add your own stop words if you want.

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer(max_features = 3000,
                       tokenizer = tokenizer_better,
                       stop_words = my_stop_words)
# this could take a while
vect.fit(df_train.TEXT.values)
```

Now we can transform our notes into numerical matrices. At this point, I will only use the training and validation data so I'm not tempted to see how it works on the test data yet.

```
X_train_tf = vect.transform(df_train.TEXT.values)
X_valid_tf = vect.transform(df_valid.TEXT.values)
```

We also need our output labels as separate variables

```
y_train = df_train.OUTPUT_LABEL
y_valid = df_valid.OUTPUT_LABEL
```

As seen by the location of the scroll bar… as always, it takes 80% of the time to get the data ready for the predictive model.

Step 3: Build a simple predictive model

We can now build a simple predictive model that takes our bag-of-words inputs and predicts if a patient will be readmitted in 30 days (YES = 1, NO = 0).

```
clf=LogisticRegression(C = 0.0001, penalty = 'l2', random_state = 42)
clf.fit(X_train_tf, y_train)
```

We can calculate the probability of readmission for each sample with the fitted model

```
model = clf
y_train_preds = model.predict_proba(X_train_tf)[:,1]
y_valid_preds = model.predict_proba(X_valid_tf)[:,1]
```

Step 4: Assess the quality of your model

At this point, we need to measure how well our model performed. There are a few different data science performance metrics. I wrote another blog post explaining these in detail if you are interested. Since this post is quite long now, I will start just showing results and figures. You can see the github account for the code to produce the figures.

With the current selection of hyperparameters, we do have some overfitting. One thing to point out is that the major difference between the precision in the two sets of data is due to the fact that we balanced the training set, where as the validation set is the original distribution. Currently, if we make a list of patients predicted to be readmitted we catch twice as many of them as if we randomly selected patients (PRECISION vs PREVALENCE).

Another performance metric not shown above is AUC or area under the ROC curve. The ROC curve for our current model is shown below. Essentially the ROC curve allows you to see the trade-off between true positive rate and false positive rate as you vary the threshold on what you define as predicted positive vs predicted negative.

Step 5: Next steps for improving the model

At this point, you might be tempted to calculate the performance on your test set and see how you did. But wait! We made many choices (a few below) which we could change and see if there is an improvement:

When I am trying to improve my models, I read a lot of other blog posts and articles to see how people tackled similar issues. When you do this, you start to see interesting ways to visualize data and I highly recommend holding on to these techniques for your own projects.

For NLP projects that use BOW+logistic regression, we can plot the most important words to see if we can gain any insight. For this step, I borrowed code from a nice NLP article by Insight Data Science. When you look at the most important words, I see two immediate things:

- Oops! I forgot to exclude the patients who died since 'expired' showed up in the negative list. For now, I will ignore this and fix it below.
- There are also some other stop words we should probably remove ('should','if','it','been','who','during', 'x')

When we want to improve the model, we want to do it in a data-driven manner. You can spend a lot of time on 'hunches', that don't end up panning out. To do this, it is recommend to pick a single performance metric that you use to make your decisions. For this project, I am going to pick AUC.

For the first question above, we can plot something called a Learning Curve to understand the effect of adding more data. Andrew Ng has a set of great Coursera classes on the discussion of high-bias vs high-variance models.

Some simple things that we can do is try to see the effect of some of our hyperparameters (`max_features` and `C`). We could run a grid search, but since we only have 2 parameters here we could look at them separately and see the effect.

Effect of C

Effect of max_features

We can see that increasing C and max_features, cause the model to overfit pretty quickly.

- concatenate all the notes instead of the last discharge summary
- try a deep learning method such as LSTMs
- review the discharge summaries that you are getting wrong

Step 6: Finalize your model and test it

We will now fit our final model with hyperparameter selection. We will also exclude the patients who died with a re-balancing.

```
rows_not_death = df_adm_notes_clean.DEATHTIME.isnull()

df_adm_notes_not_death =
df_adm_notes_clean.loc[rows_not_death].copy()
df_adm_notes_not_death = df_adm_notes_not_death.sample(n =
len(df_adm_notes_not_death), random_state = 42)
df_adm_notes_not_death = df_adm_notes_not_death.reset_index(drop =
True)

# Save 30% of the data as validation and test data
df_valid_test=df_adm_notes_not_death.sample(frac=0.30,random_state=42
)

df_test = df_valid_test.sample(frac = 0.5, random_state = 42)
df_valid = df_valid_test.drop(df_test.index)

# use the rest of the data as training data
df_train_all=df_adm_notes_not_death.drop(df_valid_test.index)

assert len(df_adm_notes_not_death) ==
(len(df_test)+len(df_valid)+len(df_train_all)),'math didnt work'

# split the training data into positive and negative
rows_pos = df_train_all.OUTPUT_LABEL == 1
df_train_pos = df_train_all.loc[rows_pos]
df_train_neg = df_train_all.loc[~rows_pos]

# merge the balanced data
```

```
['the','and','to','of','was','with','a','on','in','for','name',
'is','patient','s','he','at','as','or','one','she','his','her','am',
'were','you','pt','pm','by','be','had','your','this','date',
'from','there','an','that','p','are','have','has','h','but','o',
'namepattern','which','every','also','should','if','it','been','who',
'during', 'x']

from sklearn.feature_extraction.text import CountVectorizer

vect = CountVectorizer(lowercase = True, max_features = 3000,
                       tokenizer = tokenizer_better,
                      stop_words = my_new_stop_words)

# fit the vectorizer
vect.fit(df_train.TEXT.values)

X_train_tf = vect.transform(df_train.TEXT.values)
X_valid_tf = vect.transform(df_valid.TEXT.values)
X_test_tf = vect.transform(df_test.TEXT.values)

y_train = df_train.OUTPUT_LABEL
y_valid = df_valid.OUTPUT_LABEL
y_test = df_test.OUTPUT_LABEL

from sklearn.linear_model import LogisticRegression

clf=LogisticRegression(C = 0.0001, penalty = 'l2', random_state = 42)
clf.fit(X_train_tf, y_train)

model = clf
y_train_preds = model.predict_proba(X_train_tf)[:,1]
y_valid_preds = model.predict_proba(X_valid_tf)[:,1]
y_test_preds = model.predict_proba(X_test_tf)[:,1]
```

This produces the following results and ROC curve.

Conclusion

Congratulations! You built a simple NLP model (AUC = 0.70) to predict re-admission based on hospital discharge summaries that is only slightly worse than the state-of-the-art deep learning method that uses all hospital data (AUC = 0.75). If you have any feedback, feel free to leave it below.

If you are interested in deep learning NLP in healthcare, I recommend reading the article by Erin Craig at https://arxiv.org/abs/1711.10663

References

Scalable and accurate deep learning with electronic health records. Rajkomar A, Oren E, Chen K, et al. NPJ Digital Medicine (2018). DOI: 10.1038/s41746–018–0029–1. Available at: https://www.nature.com/articles/s41746-018-0029-1

MIMIC-III, a freely accessible critical care database. Johnson AEW, Pollard TJ, Shen L, Lehman L, Feng M, Ghassemi M, Moody B, Szolovits P, Celi LA, and Mark RG. Scientific Data (2016). DOI: 10.1038/sdata.2016.35. Available at: http://www.nature.com/articles/sdata201635

Machine Learning Bag Of Words Hospital Data Science Towards Data Science