Individual Assignment 5: ML Tutorial using MIMIC III

Clinical Natural Language Processing:

Predicting Hospital Readmission with Discharge Summaries

Step1: Prepare data for a machine learning project

In this project, I will make use of the following MIMIC III tables. The main columns of interest in this table are :

```
[4] 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

```
[5] df_adm.head()
```

	row_id	subject_id	hadm_id	admittime	dischtime	deathtime	admission_type	admission_location	discharge_location	insurance	language	religion	marital_status	ethnicity	edregtime	edouttime	diagnosis	h
0	12258	10006	142345	2164-10-23 21:09:00	2164-11-01 17:15:00	NaN	EMERGENCY	EMERGENCY ROOM ADMIT	HOME HEALTH CARE	Medicare	NaN	CATHOLIC	SEPARATED	BLACK/AFRICAN AMERICAN	2164-10-23 16:43:00	2164-10-23 23:00:00	SEPSIS	
1	12263	10011	105331	2126-08-14 22:32:00	2126-08-28 18:59:00	2126-08-28 18:59:00	EMERGENCY	TRANSFER FROM HOSP/EXTRAM	DEAD/EXPIRED	Private	NaN	CATHOLIC	SINGLE	UNKNOWN/NOT SPECIFIED	NaN	NaN	HEPATITIS B	
2	12265	10013	165520	2125-10-04 23:36:00	2125-10-07 15:13:00	2125-10-07 15:13:00	EMERGENCY	TRANSFER FROM HOSP/EXTRAM	DEAD/EXPIRED	Medicare	NaN	CATHOLIC	NaN	UNKNOWN/NOT SPECIFIED	NaN	NaN	SEPSIS	
3	12269	10017	199207	2149-05-26 17:19:00	2149-06-03 18:42:00	NaN	EMERGENCY	EMERGENCY ROOM ADMIT	SNF	Medicare	NaN	CATHOLIC	DIVORCED	WHITE	2149-05-26 12:08:00	2149-05-26 19:45:00	HUMERAL FRACTURE	
4	12270	10019	177759	2163-05-14 20:43:00	2163-05-15 12:00:00	2163-05-15 12:00:00	EMERGENCY	TRANSFER FROM HOSP/EXTRAM	DEAD/EXPIRED	Medicare	NaN	CATHOLIC	DIVORCED	WHITE	NaN	NaN	ALCOHOLIC HEPATITIS	

```
[6] df_adm.columns
```

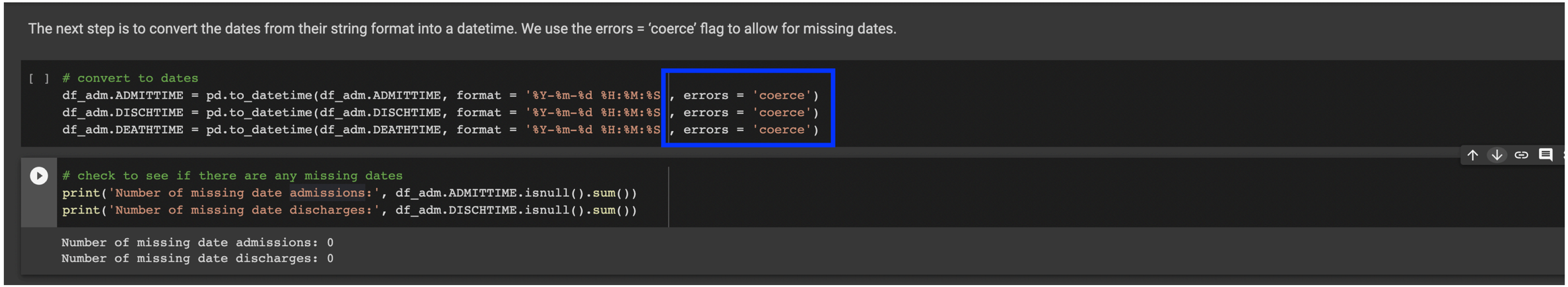
```
Index(['row_id', 'subject_id', 'hadm_id', 'admittime', 'dischtime',
       'deathtime', 'admission_type', 'admission_location',
       'discharge_location', 'insurance', 'language', 'religion',
       'marital_status', 'ethnicity', 'edregtime', 'edouttime', 'diagnosis',
       'hospital_expire_flag', 'has_chartevents_data'],
      dtype='object')
```

Admissions table: convert string to date

The next step is to convert the dates from their string format into a DateTime. I use the errors = ‘coerce’ flag to allow for missing dates.
Errors = 'coerce' allows NaT (not a datetime) to happen when the string doesn't match the format.

A note about dates from MIMIC website:

All dates in the database have been shifted to protect patient confidentiality. Dates will be internally consistent for the same patient, but randomly distributed in the future.



Get the next admission date if it exists

The next step is to get the next unplanned admission date if it exists. This will follow a few steps, and I'll show you what happens for an artificial patient. First I will sort the dataframe by the admission date

```
# sort by subject_ID and admission date
df_adm = df_adm.sort_values(['SUBJECT_ID','ADMITTIME'])
df_adm = df_adm.reset_index(drop = True)
df_adm.head()
```

	ROW_ID	SUBJECT_ID	HADM_ID	ADMITTIME	DISCHTIME	DEATHTIME	ADMISSION_TYPE	ADMISSION_LOCATION	DISCHARGE_LOCATION	INSURANCE	LANGUAGE	RELIGION	MARITAL_STATUS	ETHNICITY	EDREGTIME	EDOUTTIME	
0	1	2	163353	2138-07-17 19:04:00	2138-07-21 15:48:00	NaT	NEWBORN	PHYS REFERRAL/NORMAL DELI	HOME	Private	NaN	NOT SPECIFIED	NaN	ASIAN	NaN	NaN	
1	2	3	145834	2101-10-20 19:08:00	2101-10-31 13:58:00	NaT	EMERGENCY	EMERGENCY ROOM ADMIT	SNF	Medicare	NaN	CATHOLIC	MARRIED	WHITE	2101-10-20 17:09:00	2101-10-20 19:24:00	HY
2	3	4	185777	2191-03-16 00:28:00	2191-03-23 18:41:00	NaT	EMERGENCY	EMERGENCY ROOM ADMIT	HOME WITH HOME IV PROVIDR	Private	NaN	PROTESTANT QUAKER	SINGLE	WHITE	2191-03-15 13:10:00	2191-03-16 01:10:00	FEVER,DEHYDRAT
3	4	5	178980	2103-02-02 04:31:00	2103-02-04 12:15:00	NaT	NEWBORN	PHYS REFERRAL/NORMAL DELI	HOME	Private	NaN	BUDDHIST	NaN	ASIAN	NaN	NaN	
4	5	6	107064	2175-05-30 07:15:00	2175-06-15 16:00:00	NaT	ELECTIVE	PHYS REFERRAL/NORMAL DELI	HOME HEALTH CARE	Medicare	ENGL	NOT SPECIFIED	MARRIED	WHITE	NaN	NaN	CHR

```
df_adm.groupby(['ADMISSION_TYPE']).size()
```

```
ADMISSION_TYPE
ELECTIVE      7706
EMERGENCY    42071
NEWBORN       7863
URGENT        1336
dtype: int64
```

Get the next admission date if it exists

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

```
[ ] # verify that it did what we wanted
    df_adm.loc[df_adm.SUBJECT_ID == 124,['SUBJECT_ID','ADMITTIME','ADMISSION_TYPE']]
```

	SUBJECT_ID	ADMITTIME	ADMISSION_TYPE
165	124	2160-06-24 21:25:00	EMERGENCY
166	124	2161-12-17 03:39:00	EMERGENCY
167	124	2165-05-21 21:02:00	ELECTIVE
168	124	2165-12-31 18:55:00	EMERGENCY

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)
```

```
[ ] # verify that it did what we wanted
    df_adm.loc[df_adm.SUBJECT_ID == 124,['SUBJECT_ID','ADMITTIME','ADMISSION_TYPE','NEXT_ADMITTIME','NEXT_ADMISSION_TYPE']]
```

	SUBJECT_ID	ADMITTIME	ADMISSION_TYPE	NEXT_ADMITTIME	NEXT_ADMISSION_TYPE
165	124	2160-06-24 21:25:00	EMERGENCY	2161-12-17 03:39:00	EMERGENCY
166	124	2161-12-17 03:39:00	EMERGENCY	2165-05-21 21:02:00	ELECTIVE
167	124	2165-05-21 21:02:00	ELECTIVE	2165-12-31 18:55:00	EMERGENCY
168	124	2165-12-31 18:55:00	EMERGENCY	NaT	NaN

Get the next admission date if it exists

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

```
[ ]  # get rows where next admission is elective and replace with naT or nan
     rows = df_adm.NEXT_ADMISSION_TYPE == 'ELECTIVE'
     df_adm.loc[rows,'NEXT_ADMITTIME'] = pd.NaT
     df_adm.loc[rows,'NEXT_ADMISSION_TYPE'] = np.NaN
```

```
# verify that it did what we wanted
df_adm.loc[df_adm.SUBJECT_ID == 124,['SUBJECT_ID','ADMITTIME','ADMISSION_TYPE','NEXT_ADMITTIME','NEXT_ADMISSION_TYPE']]
```

	SUBJECT_ID	ADMITTIME	ADMISSION_TYPE	NEXT_ADMITTIME	NEXT_ADMISSION_TYPE
165	124	2160-06-24 21:25:00	EMERGENCY	2161-12-17 03:39:00	EMERGENCY
166	124	2161-12-17 03:39:00	EMERGENCY	NaT	NaN
167	124	2165-05-21 21:02:00	ELECTIVE	2165-12-31 18:55:00	EMERGENCY
168	124	2165-12-31 18:55:00	EMERGENCY	NaT	NaN

We can then calculate the days until the next admission

```
[ ]  # sort by subject_ID and admission date
     # it is safer to sort right before the fill incase 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')
```

```
[ ]  # verify that it did what we wanted
     df_adm.loc[df_adm.SUBJECT_ID == 124,['SUBJECT_ID','ADMITTIME','ADMISSION_TYPE','NEXT_ADMITTIME','NEXT_ADMISSION_TYPE']]
```

	SUBJECT_ID	ADMITTIME	ADMISSION_TYPE	NEXT_ADMITTIME	NEXT_ADMISSION_TYPE
165	124	2160-06-24 21:25:00	EMERGENCY	2161-12-17 03:39:00	EMERGENCY
166	124	2161-12-17 03:39:00	EMERGENCY	2165-12-31 18:55:00	EMERGENCY
167	124	2165-05-21 21:02:00	ELECTIVE	2165-12-31 18:55:00	EMERGENCY

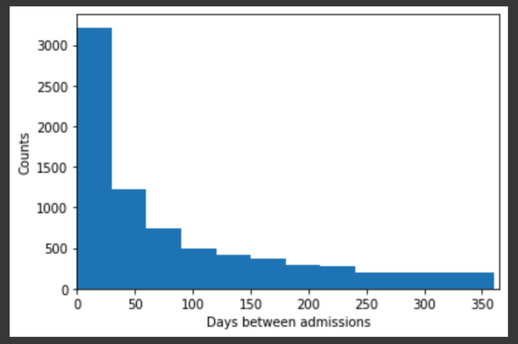
Calculate days until next admission

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.

```
[ ] # calculate the number of days between discharge and 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.

```
# plot a histogram of days between readmissions if they exist
# this only works for non-null values so you have to filter
plt.hist(df_adm.loc[~df_adm.DAYS_NEXT_ADMIT.isnull(),'DAYS_NEXT_ADMIT'], bins =range(0,365,30))
plt.xlim([0,365])
plt.xlabel('Days between admissions')
plt.ylabel('Counts')
plt.show()
```




```
print('Number with a readmission:', (~df_adm.DAYS_NEXT_ADMIT.isnull()).sum())
print('Total Number:', len(df_adm))
```

```
Number with a readmission: 11399
Total Number: 58976
```

Merge data sets

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. Additional analysis might be required to see if there are any specific cases why a discharge would not appear.

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'
```

```
[ ] len(df_adm)

58976
```

```
[ ] len(df_adm_notes)

58976
```

```
[ ] df_adm_notes.TEXT.isnull().sum()

6250
```

```
[ ]
print('Fraction of missing notes:', df_adm_notes.TEXT.isnull().sum() / len(df_adm_notes))
print('Fraction notes with newlines:', df_adm_notes.TEXT.str.contains('\n').sum() / len(df_adm_notes))
print('Fraction notes with carriage returns:', df_adm_notes.TEXT.str.contains('\r').sum() / len(df_adm_notes))

Fraction of missing notes: 0.1059753119913185
Fraction notes with newlines: 0.8940246880086815
Fraction notes with carriage returns: 0.0
```

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()

ADMISSION_TYPE
ELECTIVE     0.048663
EMERGENCY    0.037983
NEWBORN      0.536691
URGENT       0.042665
dtype: float64
```

Prepare a label and creating training test dataframes

I like to create a specific column in the data frame as OUTPUT_LABEL that has exactly what we are trying to predict. When I build a Predictive model, I take data and split it into three datasets: training, validation, and test.

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

```
[ ]
    print('Number of positive samples:', (df_adm_notes_clean.OUTPUT_LABEL == 1).sum())
    print('Number of negative samples:',  (df_adm_notes_clean.OUTPUT_LABEL == 0).sum())
    print('Total samples:', len(df_adm_notes_clean))
```

```
Number of positive samples: 3004
Number of negative samples: 48109
Total samples: 51113
```

```
# 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)

print('Test prevalence(n = %d):'%len(df_test),df_test.OUTPUT_LABEL.sum()/ len(df_test))
print('Valid prevalence(n = %d):'%len(df_valid),df_valid.OUTPUT_LABEL.sum()/ len(df_valid))
print('Train all prevalence(n = %d):'%len(df_train_all), df_train_all.OUTPUT_LABEL.sum()/ len(df_train_all))
print('all samples (n = %d)'%len(df_adm_notes_clean))
assert len(df_adm_notes_clean) == (len(df_test)+len(df_valid)+len(df_train_all)),'math didnt work'
```

```
Test prevalence(n = 7667): 0.061953828094430674
Valid prevalence(n = 7667): 0.056997521846876224
Train all prevalence(n = 35779): 0.05847005226529529
all samples (n = 51113)
```

```
[ ] # 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
    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)

    print('Train prevalence (n = %d):'%len(df_train), df_train.OUTPUT_LABEL.sum()/ len(df_train))
```

```
Train prevalence (n = 4184): 0.5
```

- **training set:** used to train the model
- **validation set:** data the model didn't see but are used to optimize or tune the model
- **test set:** data the model and tuning process never saw (the true test of generalizability)

Step2: Preprocess text data

Modify the original data frame by dealing with the missing text, newlines and carriage returns. Now that I 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

```
[ ]
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
```

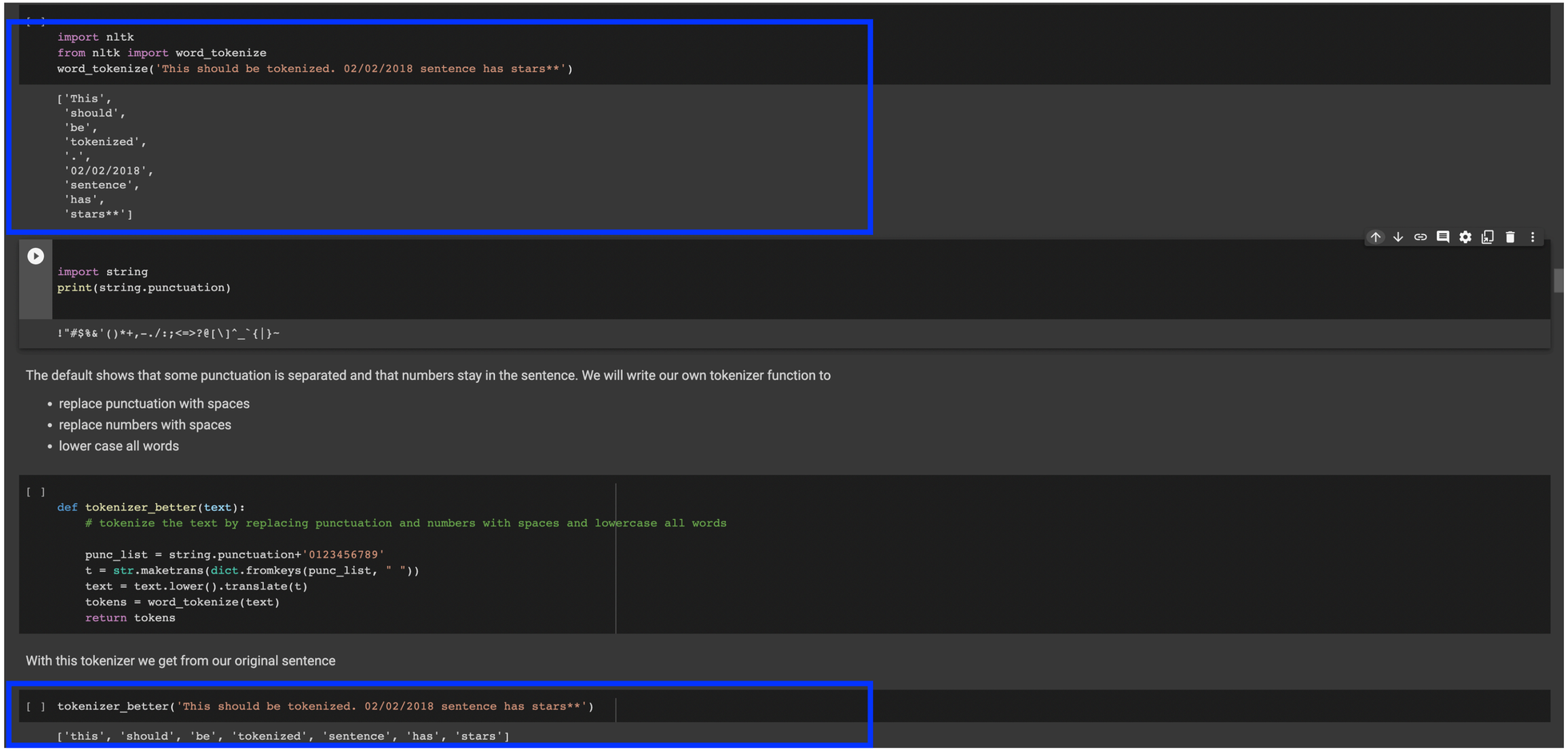
```
[ ] # preprocess the text to deal with known issues
df_train = preprocess_text(df_train)
df_valid = preprocess_text(df_valid)
df_test = preprocess_text(df_test)
```

```
[ ] nltk.download('all')
```

```
[nltk_data] Downloading collection 'all'
[nltk_data]    |
[nltk_data]    | Downloading package abc to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/abc.zip.
[nltk_data]    | Downloading package alpino to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/alpino.zip.
[nltk_data]    | Downloading package biocreative_ppi to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping corpora/biocreative_ppi.zip.
[nltk_data]    | Downloading package brown to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/brown.zip.
[nltk_data]    | Downloading package brown_tei to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/brown_tei.zip.
[nltk_data]    | Downloading package cess_cat to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/cess_cat.zip.
[nltk_data]    | Downloading package cess_esp to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/cess_esp.zip.
[nltk_data]    | Downloading package chat80 to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/chat80.zip.
[nltk_data]    | Downloading package city_database to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping corpora/city_database.zip.
[nltk_data]    | Downloading package cmudict to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/cmudict.zip.
[nltk_data]    | Downloading package comparative_sentences to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping corpora/comparative_sentences.zip.
[nltk_data]    | Downloading package comtrans to /root/nltk_data...
[nltk_data]    | Downloading package conll2000 to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/conll2000.zip.
[nltk_data]    | Downloading package conll2002 to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/conll2002.zip.
[nltk_data]    | Downloading package conll2007 to /root/nltk_data...
[nltk_data]    | Downloading package crubadan to /root/nltk_data...
```

Build a tokenizer

This process consists of using a tokenizer and a vectorizer. The tokenizer breaks a single note into a list of words and a vectorizer takes a list of words and counts the words. I will use word_tokenize from the nltk package for our default tokenizer, which basically breaks the note based on spaces and some punctuation.



```
[ ]
import nltk
from nltk import word_tokenize
word_tokenize('This should be tokenized. 02/02/2018 sentence has stars**')
```

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

```
import string
print(string.punctuation)
```

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

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

```
[ ]
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
```

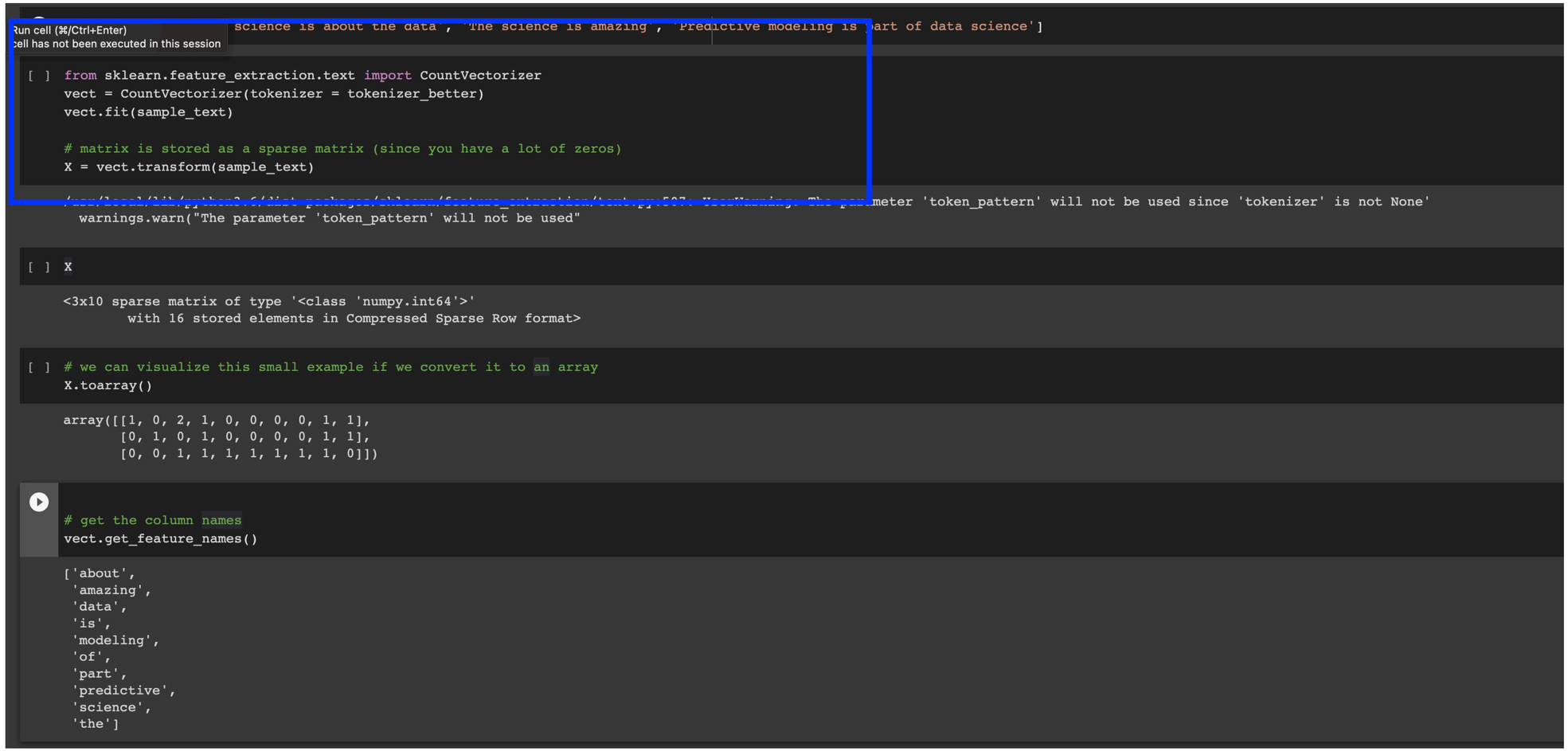
With this tokenizer we get from our original sentence

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

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

Build a simple vectorizer

Now that we have a way to convert free text into tokens, I need a way to count the tokens for each discharge summary. I will use the built-in CountVectorizer from the scikit-learn package. This vectorizer simply counts how many times each word occurs in the note. There is also a TfidfVectorizer that takes into how often words are used across all notes, but for this project let's use the simpler one



Build a vectorizer on the clinical notes

I can now fit our CountVectorizer on the clinical notes. It is important to use only the training data because I don't want to include any new words that show up in the validation and test sets. There is a hyperparameter called max_features which I can set to constrain how many words are included in the Vectorizer. This will use the top N most frequently used words.

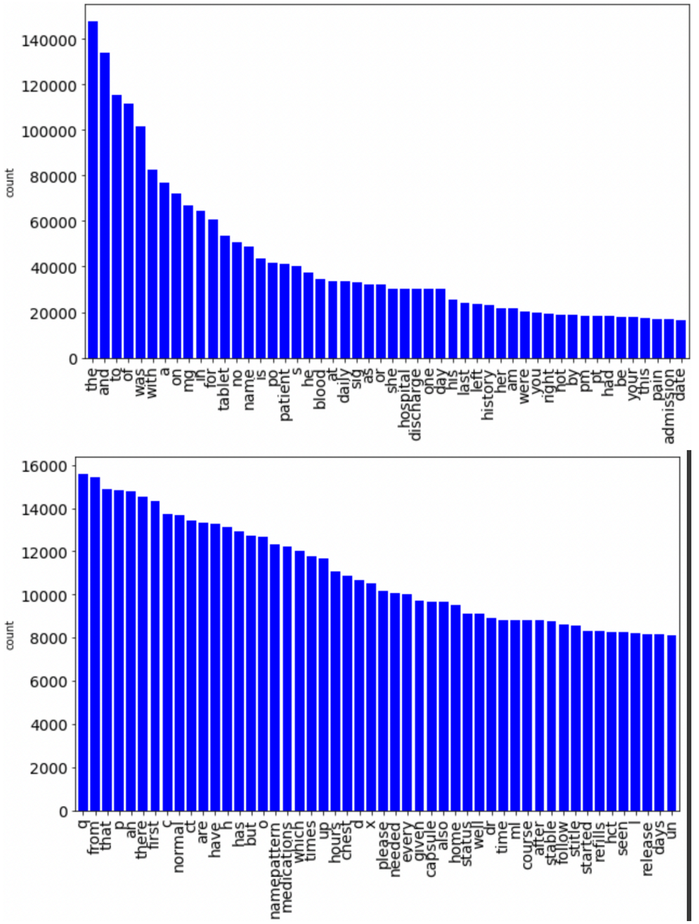
```
[ ]
    from sklearn.feature_extraction.text import CountVectorizer
    vect = CountVectorizer(max_features = 3000, tokenizer = tokenizer_better)

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

```
    /usr/local/lib/python3.6/dist-packages/sklearn/feature_extraction/text.py:507: UserWarning: The parameter 'token_pattern' will not be used since 'tokenizer' is not None'
      warnings.warn("The parameter 'token_pattern' will not be used"
    CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                    dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                    lowercase=True, max_df=1.0, max_features=3000, min_df=1,
                    ngram_range=(1, 1), preprocessor=None, stop_words=None,
                    strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                    tokenizer=<function tokenizer_better at 0x7efcf7dd9158>,
                    vocabulary=None)
```

Zipf's law

I 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 I can remove them easily (if we want) with the CountVectorizer. There are lists of common stop words for different NLP corpus, but I will just make up our own based on the image below.

```
neg_doc_matrix = vect.transform(df_train[df_train.OUTPUT_LABEL == 0].TEXT)
pos_doc_matrix = vect.transform(df_train[df_train.OUTPUT_LABEL == 1].TEXT)
neg_tf = np.sum(neg_doc_matrix,axis=0)
pos_tf = np.sum(pos_doc_matrix,axis=0)
neg = np.squeeze(np.asarray(neg_tf))
pos = np.squeeze(np.asarray(pos_tf))

term_freq_df = pd.DataFrame([neg,pos],columns=vect.get_feature_names()).transpose()
term_freq_df.columns = ['negative', 'positive']
term_freq_df['total'] = term_freq_df['negative'] + term_freq_df['positive']
term_freq_df.sort_values(by='total', ascending=False).iloc[:10]

#Create a series from the sparse matrix
d = pd.Series(term_freq_df.total,
              index = term_freq_df.index).sort_values(ascending=False)
ax = d[:50].plot(kind='bar', figsize=(10,6), width=.8, fontsize=14, rot=90,color = 'b')
ax.title.set_size(18)
plt.ylabel('count')
plt.show()
ax = d[50:100].plot(kind='bar', figsize=(10,6), width=.8, fontsize=14, rot=90,color = 'b')
ax.title.set_size(18)
plt.ylabel('count')
plt.show()
```



Step3: Build a simple predictive

I 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). Here I will use the Logistic Regression model. Logistic regression is a good baseline model for NLP tasks since it works well with sparse matrices and is interpretable.

```
[ ] # logistic regression
    from sklearn.linear_model import LogisticRegression
    clf=LogisticRegression(C = 0.0001, penalty = 'l2', random_state = 42)
    clf.fit(X_train_tf, y_train)
```

```
    LogisticRegression(C=0.0001, class_weight=None, dual=False, fit_intercept=True,
                       intercept_scaling=1, l1_ratio=None, max_iter=100,
                       multi_class='auto', n_jobs=None, penalty='l2',
                       random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                       warm_start=False)
```

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]
```

Step4: Assess the quality of your model

At this point, I 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 by showing results and figures. You can see the GitHub account for the code to produce the figures.

```
[ ]  print(y_train[:10].values)
     print(y_train_preds[:10])
```

```
[1 1 0 1 1 1 0 0 1 1]
[0.42013987 0.4360379  0.28809073 0.6238073  0.24422986 0.45490439
 0.33683096 0.91645687 0.69387608 0.50733574]
```

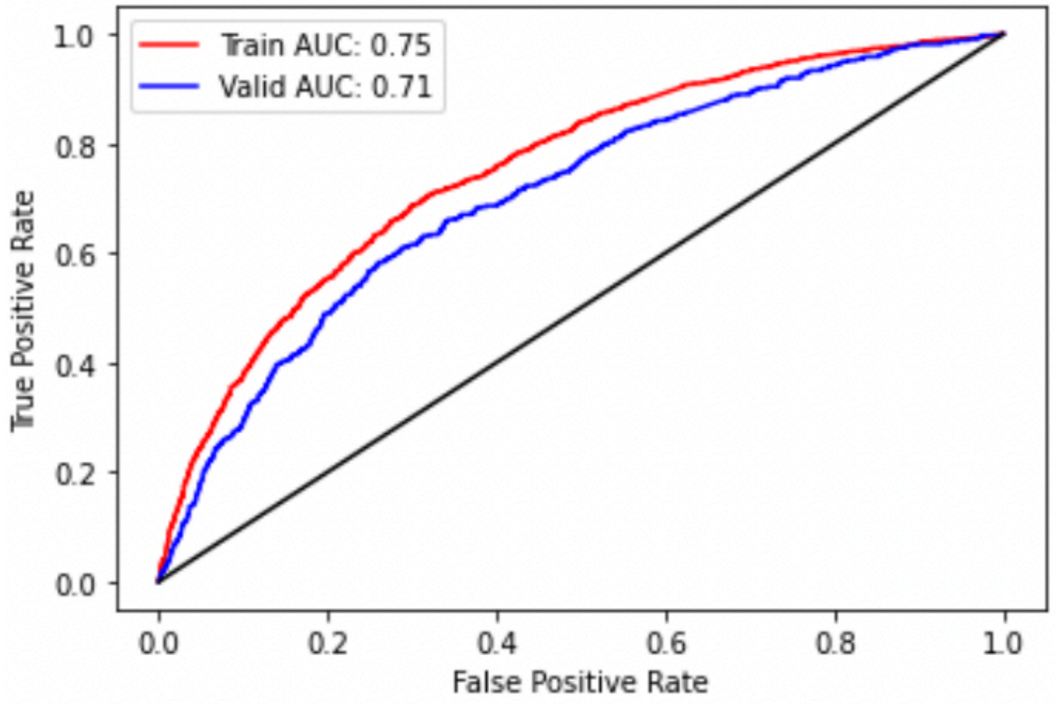
```
def calc_accuracy(y_actual, y_pred, thresh):
    # this function calculates the accuracy with probability threshold at thresh
    return (sum((y_pred > thresh) & (y_actual == 1))+sum((y_pred < thresh) & (y_actual == 0))) /len(y_actual)

def calc_recall(y_actual, y_pred, thresh):
    # calculates the recall
    return sum((y_pred > thresh) & (y_actual == 1)) /sum(y_actual)

def calc_precision(y_actual, y_pred, thresh):
    # calculates the precision
    return sum((y_pred > thresh) & (y_actual == 1)) /sum(y_pred > thresh)

def calc_specificity(y_actual, y_pred, thresh):
    # calculates specificity
    return sum((y_pred < thresh) & (y_actual == 0)) /sum(y_actual ==0)

def calc_prevalence(y_actual):
    # calculates prevalence
    return sum((y_actual == 1)) /len(y_actual)
```

Step4: Assess the quality of your model

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.

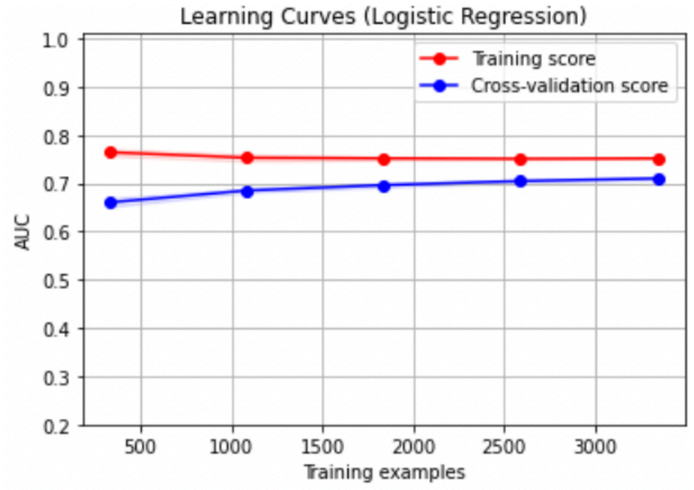
```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

fpr_train, tpr_train, thresholds_train = roc_curve(y_train, y_train_preds)
fpr_valid, tpr_valid, thresholds_valid = roc_curve(y_valid, y_valid_preds)


thresh = 0.5

auc_train = roc_auc_score(y_train, y_train_preds)
auc_valid = roc_auc_score(y_valid, y_valid_preds)

print('Train AUC:%.3f'%auc_train)
print('Valid AUC:%.3f'%auc_valid)

print('Train accuracy:%.3f'%calc_accuracy(y_train, y_train_preds, thresh))
print('Valid accuracy:%.3f'%calc_accuracy(y_valid, y_valid_preds, thresh))


print('Train recall:%.3f'%calc_recall(y_train, y_train_preds, thresh))
print('Valid recall:%.3f'%calc_recall(y_valid, y_valid_preds, thresh))

print('Train precision:%.3f'%calc_precision(y_train, y_train_preds, thresh))
print('Valid precision:%.3f'%calc_precision(y_valid, y_valid_preds, thresh))

print('Train specificity:%.3f'%calc_specificity(y_train, y_train_preds, thresh))
print('Valid specificity:%.3f'%calc_specificity(y_valid, y_valid_preds, thresh))

print('Train prevalence:%.3f'%calc_prevalence(y_train))
print('Valid prevalence:%.3f'%calc_prevalence(y_valid))


plt.plot(fpr_train, tpr_train,'r-', label = 'Train AUC: %.2f'%auc_train)
plt.plot(fpr_valid, tpr_valid,'b-',label = 'Valid AUC: %.2f'%auc_valid)
plt.plot([0,1],[0,1],'-k')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```

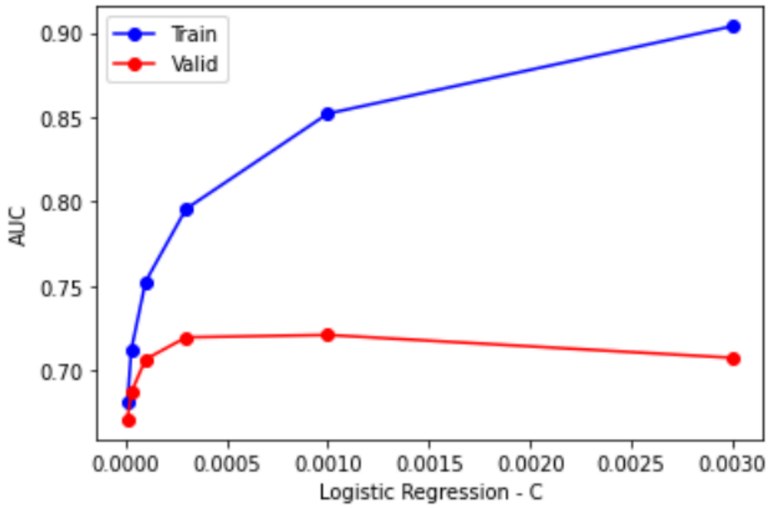
```
Train AUC:0.754
Valid AUC:0.706
Train accuracy:0.685
Valid accuracy:0.714
Train recall:0.624
Valid recall:0.595
Train precision:0.712
Valid precision:0.114
Train specificity:0.747
Valid specificity:0.721
Train prevalence:0.500
Valid prevalence:0.057
```

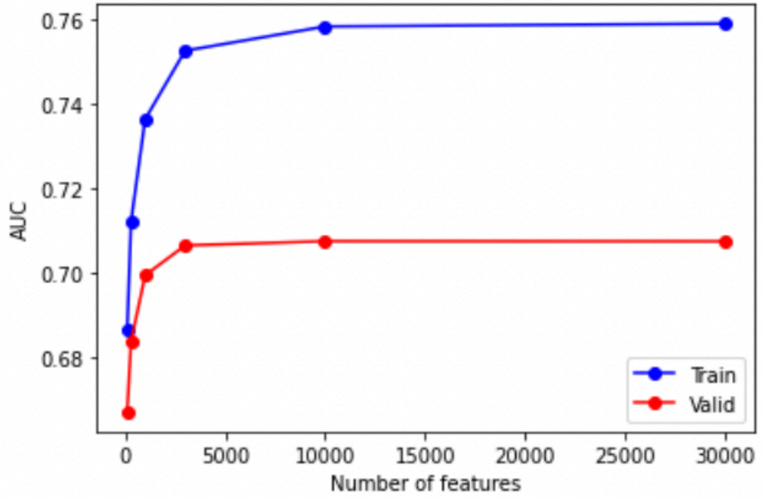


Step5: Try to improve the model: plot a learning curve

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







This is good to know because it means we shouldn't spend months getting more data. Some simple thing that we can do is try to see the effect of some of our hyperparameters (max_features and C).

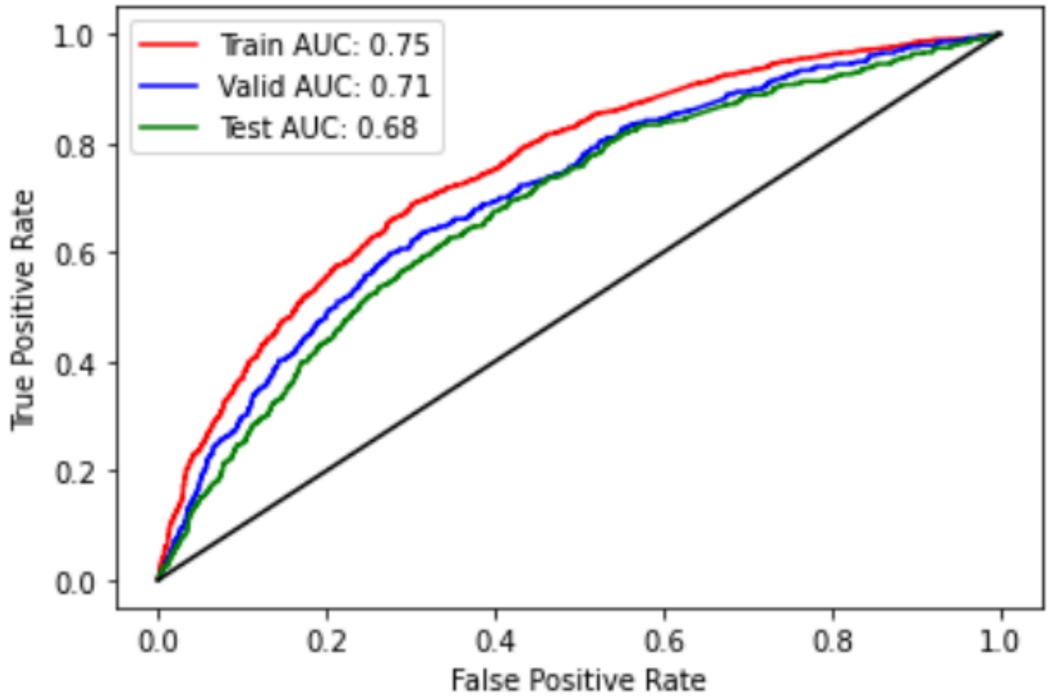
I could run a grid search, but since I only have 2 parameters here we could look at them separately and see the effect.

I can see that increasing C and max_features, cause the model to overfit pretty quickly. I selected C = 0.0001 and max_features = 3000 where the validation set started to plateau. At this point, you could try a few other things

Step6: Finalize your model and test it

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

Conventionally, I follow the tutorial to build a simple NLP model (AUC = 0.70) to predict re-admission based on hospital discharge summaries which is only slightly worse than the state-of-the-art deep learning method that uses all hospital data (AUC = 0.75).



rain	prevalence(n	=	4184):		0.500
Valid	prevalence(n	=	7667)	:	0.057
Test	prevalence(n	=	7667):		0.062
Train	AUC:0.753				
Valid	AUC:0.707				
Test	AUC:0.683				
Train	accuracy:0.6	86			
Valid	accuracy:0.7	13			
Test	accuracy:0.70	9			
Train	recall:0.623				
Valid	recall:0.600				
Test	recall:0.549				
Train	precision:0.	71	3		
Valid	precision:0.	11	5		
Test	precision:0.1	15			
Train	specificity:	0	749		
Valid	specificity:	0	720		
Test	specificity:0	0.7	19		

Clinical Natural Language Processing:

Predicting ICD-9 classification using PySpark SQL

Step1: Initialization and data loading

Spark ML provides a set of Machine Learning applications that can be build using two main components: Estimators and Transformers. The Estimators have a method called fit() which secures and trains a piece of data to such application. The Transformer is generally the result of a fitting process and applies changes to the the target dataset. These components have been embedded to be applicable to Spark NL

Annotators

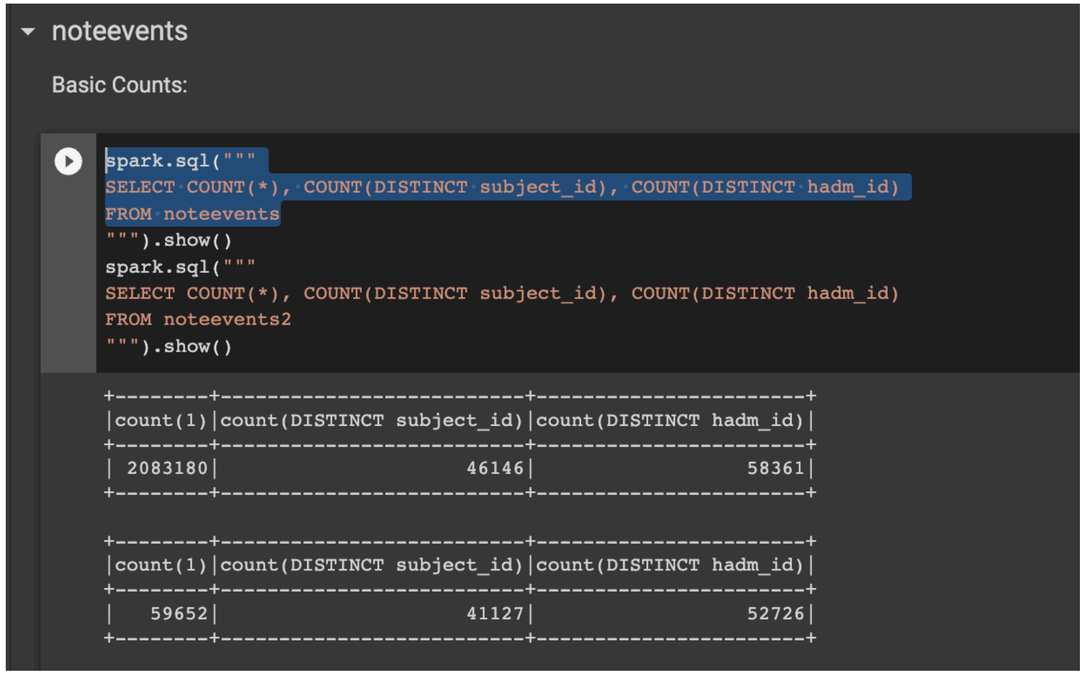
* SparkConf(): Create a SparkConf that loads defaults from system properties and the classpath
* setAppName:
* setMaster(String master) The master URL to connect to, such as "local" to run locally with one thread, "local" to run locally with 4 cores.
* SparkContext: Create a SparkContext that loads settings from system properties (for instance, when launching with ./bin/spark-submit).
* SparkSession: Unified entry point of a spark application from Spark2.0.
* StructType: Provides spark.sql.types.StructType class to define the structure of the DataFrame and It is a collection or list on StructField objects.
* StructField: Defines the metadata of the DataFrame column
* selectExpr: Select() is a transformation function that is used to select the columns from DataFrame and Dataset, It has two different types of syntaxes.

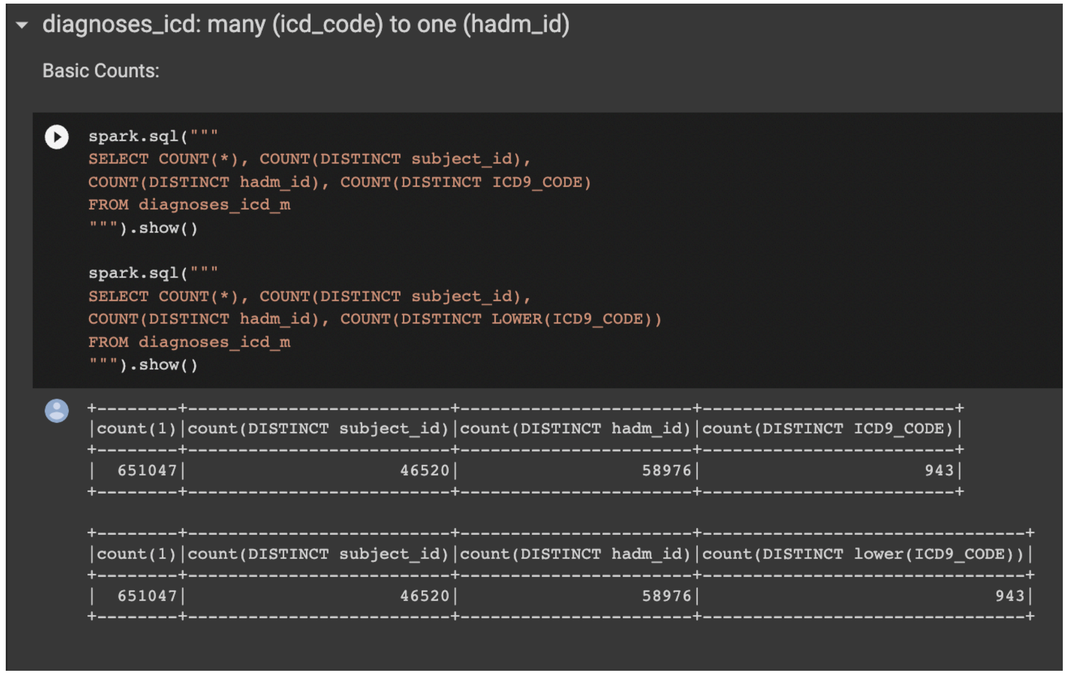
Step2: Descriptive Statistics

In this Spark SQL tutorial, you will learn different ways to count the distinct values in every column or selected columns of rows in a DataFrame using methods available on DataFrame and SQL function using Scala examples.

Using SQL Count distinct

`distinct()` runs distinct on all columns, if you want to get count distinct on selected columns, use the Spark SQL function `countDistinct()`. This function returns the number of distinct elements in a group.





Step2: Descriptive Statistics

I can select all from `icd9_code` table and save the result to DataFrame or DataSet. Then I use "limit" in this query (limit 50) and can just call the show(50) method. As **Spark SQL** does not support TOP clause thus I tried to use the syntax of MySQL which is the "**LIMIT**" clause.

Top 50 ICD 9 codes based on "subject_id" count

```
spark.sql("""
SELECT icd9_code, COUNT(DISTINCT subject_id) AS sid_count
FROM diagnoses_icd_o2
GROUP BY icd9_code
ORDER BY sid_count DESC
LIMIT 50
""").show(n=50)
```

```
+---------+---------+
|icd9_code|sid_count|
+---------+---------+
|      414|     3503|
|      410|     3137|
|      038|     2966|
|      V30|     2348|
|      424|     1691|
|      518|     1324|
|      428|     1248|
|      996|     1199|
|      V31|      981|
|      431|      948|
|      852|      903|
|      427|      900|
|      998|      726|
|      441|      724|
|      434|      690|
|      486|      654|
|      250|      631|
|      584|      611|
|      578|      606|
|      507|      583|
|      198|      513|
|      430|      491|
|      162|      455|
|      571|      427|
|      801|      419|
|      577|      394|
|      562|      377|
|      415|      363|
|      440|      361|
|      433|      342|
|      997|      324|
|      396|      311|
|      197|      308|
|      805|      278|
|      965|      277|
|      482|      277|
|      432|      268|
|      780|      265|
|      519|      260|
|      437|      254|
|      532|      251|
|      820|      244|
|      851|      243|
|      560|      240|
|      V34|      235|
|      276|      231|
|      345|      229|
|      291|      225|
|      530|      221|
|      491|      221|
+---------+---------+
```

Top 50 ICD 9 codes based on "subject_id" count

Top 50 ICD 9 codes based on "hadm_id" count

```
spark.sql("""
SELECT icd9_code, COUNT(DISTINCT hadm_id) AS hadm_count
FROM diagnoses_icd_o2
GROUP BY icd9_code
ORDER BY hadm_count DESC
LIMIT 50
""").show(n=50)
```

```
+---------+----------+
|icd9_code|hadm_count|
+---------+----------+
|      414|      3540|
|      038|      3276|
|      410|      3228|
|      V30|      2348|
|      424|      1707|
|      518|      1510|
|      428|      1460|
|      996|      1373|
|      V31|       981|
|      431|       966|
|      427|       962|
|      852|       940|
|      250|       884|
|      441|       782|
|      998|       747|
|      486|       703|
|      434|       693|
|      578|       656|
|      507|       643|
|      584|       634|
|      198|       553|
|      430|       495|
|      571|       483|
|      162|       471|
|      577|       434|
|      801|       419|
|      562|       404|
|      440|       389|
|      415|       367|
|      433|       353|
|      997|       332|
|      197|       328|
|      519|       320|
|      396|       314|
|      291|       314|
|      437|       296|
|      482|       293|
|      432|       285|
|      491|       284|
|      805|       281|
|      965|       281|
|      780|       268|
|      532|       257|
|      560|       254|
|      345|       253|
|      851|       245|
|      820|       244|
|      276|       241|
|      V34|       235|
|      530|       234|
+---------+----------+
```

Top 50 ICD 9 codes based on "hadm_id" count

Step2: Descriptive Statistics

A select distinct statement first builds the overall result set with all records, including `JOIN` statements. The select distinct statement is used to return only distinct (different) values. Inside the table, a column contains multiple duplicate values. You can **use CTE to get the distinct values of the second table, and then join that with the first table**. You also need to get the distinct values based on LastName column. You do this with a Row_Number() partitioned by the LastName, and sorted by the FirstName

```
[ ] spark.sql("""
    SELECT COUNT(DISTINCT subject_id),
    COUNT(DISTINCT hadm_id), COUNT(DISTINCT icd9_code)
    FROM diagnoses_icd_m2
    """).show()

    spark.sql("""
    SELECT COUNT(DISTINCT subject_id),
    COUNT(DISTINCT hadm_id), COUNT(DISTINCT icd9_code)
    FROM (
        SELECT row_id, subject_id, diagnoses_icd_m.hadm_id AS hadm_id,
        seq_num, icd9_code
        FROM diagnoses_icd_m JOIN (SELECT DISTINCT hadm_id FROM noteevents) AS a
        ON diagnoses_icd_m.hadm_id = a.hadm_id
    )
    """).show()
```

```
+------------------------+----------------------+------------------------+
|count(DISTINCT subject_id)|count(DISTINCT hadm_id)|count(DISTINCT icd9_code)|
+------------------------+----------------------+------------------------+
|                   41127|                 52726|                     942|
+------------------------+----------------------+------------------------+

+------------------------+----------------------+------------------------+
|count(DISTINCT subject_id)|count(DISTINCT hadm_id)|count(DISTINCT icd9_code)|
+------------------------+----------------------+------------------------+
|                   46139|                 58361|                     943|
+------------------------+----------------------+------------------------+
```