Drug reviews sentiment analysis

Background Information

A hospital or insurance provider is interested in efficiently extracting numeric ratings from patients' written review. To this end we build a model using labelled, numerically, patient reviews.

Data exploration

The data comes from Drugs.com and is accessed through UCI's website. Click [here ('.//drugsComTrain_raw.tsv')](.//drugsComTrain_raw.tsv) to access data.

In [3]:
```
import pandas as pd

```

In [4]:
```
import numpy as np

```

In [5]:
```
import matplotlib.pyplot as plt
```

In [6]:
```
tr = pd.read_table('.//drugsComTrain_raw.tsv')
```

In [7]:
```
tr.head()
```

Out[7]:

	Unnamed: 0	drugName	condition	review	rating	date	usefulCount
0	206461	Valsartan	Left Ventricular Dysfunction	"It has no side effect, I take it in combinati...	9.0	May 20, 2012	27
1	95260	Guanfacine	ADHD	"My son is halfway through his fourth week of ...	8.0	April 27, 2010	192
2	92703	Lybrel	Birth Control	"I used to take another oral contraceptive, wh...	5.0	December 14, 2009	17
3	138000	Ortho Evra	Birth Control	"This is my first time using any form of birth...	8.0	November 3, 2015	10
4	35696	Buprenorphine / naloxone	Opiate Dependence	"Suboxone has completely turned my life around...	9.0	November 27, 2016	37

There are five potential independent variables, one target variable, and a unique id column, for total of 7 columns.

In [8]:

```
1 tr.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 161297 entries, 0 to 161296
Data columns (total 7 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   Unnamed: 0   161297 non-null  int64  
 1   drugName     161297 non-null  object 
 2   condition    160398 non-null  object 
 3   review       161297 non-null  object 
 4   rating       161297 non-null  float64
 5   date         161297 non-null  object 
 6   usefulCount  161297 non-null  int64  
dtypes: float64(1), int64(2), object(4)
memory usage: 8.6+ MB
```

There are very few missing values, only 'condition' has missing values of about 1,000, less than 1% of total sample.

In [9]:

```
1 trd = tr.dropna()
```

In [10]:

```
1 trd.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 160398 entries, 0 to 161296
Data columns (total 7 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   Unnamed: 0   160398 non-null  int64  
 1   drugName     160398 non-null  object 
 2   condition    160398 non-null  object 
 3   review       160398 non-null  object 
 4   rating       160398 non-null  float64
 5   date         160398 non-null  object 
 6   usefulCount  160398 non-null  int64  
dtypes: float64(1), int64(2), object(4)
memory usage: 9.8+ MB
```

Dealing with missing values.

Removing nonsensical samples in condition by creating new, clean column.

In [11]:

```
trd['condC']= trd['condition'].map(lambda x: x if "users" not in str(x
```

```
C:\Users\jmark\AppData\Local\Temp\ipykernel_22508\1641903905.py:1: Setting
WithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-doc
s/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://
pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-
view-versus-a-copy)
  trd['condC']= trd['condition'].map(lambda x: x if "users" not in str(x)
else '')
```

In [12]:

```
 trd= trd.drop('condition', axis = 1)
```

In [13]:

```
trd.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 160398 entries, 0 to 161296
Data columns (total 7 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   Unnamed: 0   160398 non-null  int64
 1   drugName     160398 non-null  object
 2   review       160398 non-null  object
 3   rating       160398 non-null  float64
 4   date         160398 non-null  object
 5   usefulCount  160398 non-null  int64
 6   condC        160398 non-null  object
dtypes: float64(1), int64(2), object(4)
memory usage: 9.8+ MB
```

It initially appears there are 812 unique condtions, but some entries may overlap (i.e. heart failure /attack and different types of diabetes)

In [14]:

```
len(trd.condC.unique())
```

Out[14]: 812

In [15]:

```


count = 0
sets = {}
for i in trd.condC:
    if sets.get(i, 0) >0:
        sets[i]+=1
    else:
        sets[i]=1

```

In [16]:
```
len(sets.values())
```
Out[16]: 812

The data is fairly balanced by drug, uniformly distributed.

In [17]:
```


count = 0
sets = {}
for i in trd.drugName:
    if sets.get(i, 0) >0:
        sets[i]+=1
    else:
        sets[i]=1

```

In [18]:
```
len(sets)
```
Out[18]: 3431

In [19]:
```
len(set(trd.drugName.sort_values()))
```
Out[19]: 3431

It initially appears there are 3431 unique drugs, however names may overlap with eachother, and not be so distinct.

Written review metrics:

In [20]:
```
sm=0
for i in range(10):
    sm += len(trd['review'][i])


sm/11/5
```
Out[20]: 90.0

About 90 words per review, given sample of 11 and average 5 words per sentence.

Example below:

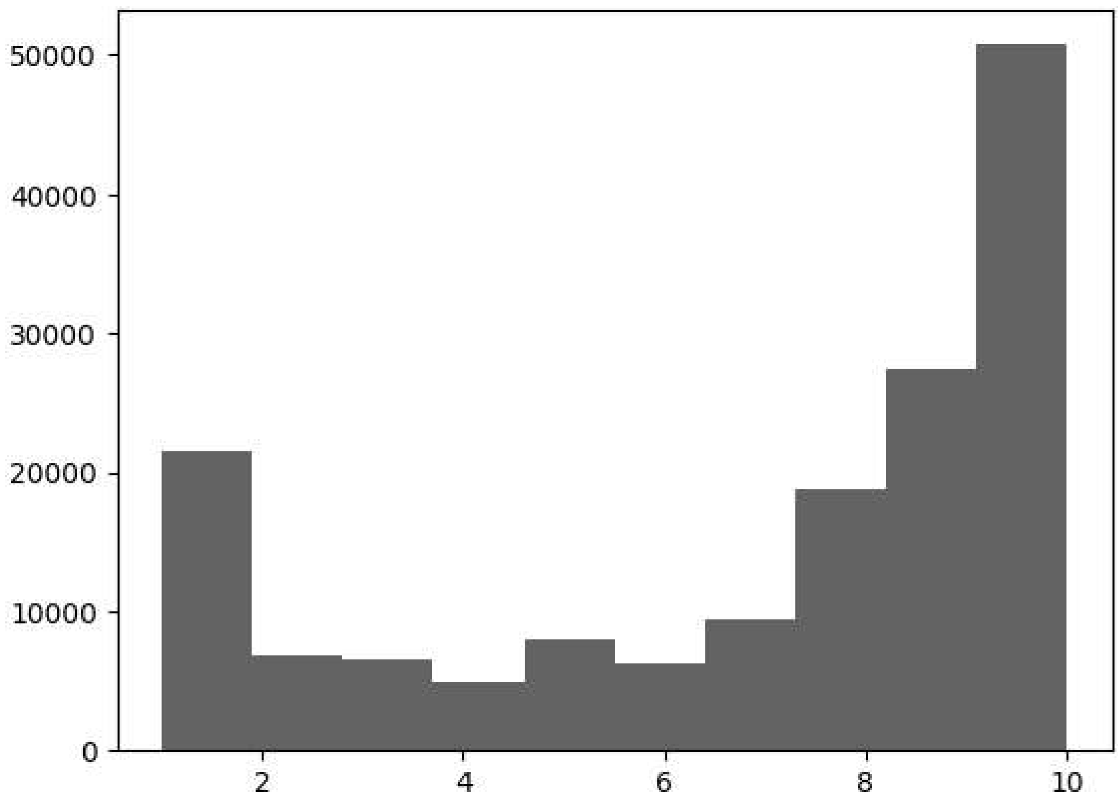
In [21]:
```
trd.review[22]
```
Out[21]:
```
'"Nexplanon does its job. I can have worry free sex. The only thing is tha
t my periods are sometimes light and sometimes heavy. Sometimes they go aw
ay and sometimes they show up unexpected. I also feel somewhat depressed.
Not sure if its Nexplanon or not. I&#039;ve had Nexplanont for about 2 mon
ths now, but despite the side effects its the most effective birth control
I&#039;ve ever used and I do not plan on taking it out."'
```

Below is a historgam of the patients' ratings.

In [22]:
```
fig, ax =plt.subplots()

ax.hist(trd['rating'])
```

Out[22]:
```
(array([21504.,  6879.,  6465.,  4980.,  7959.,  6301.,  9395., 18791.,
        27379., 50745.]),
 array([ 1. ,  1.9,  2.8,  3.7,  4.6,  5.5,  6.4,  7.3,  8.2,  9.1, 10.
]),
 <BarContainer object of 10 artists>)
```



Ratings are not normally distributed. Counts are highest at the worst and best ratings.

Data processing

Data processing involves tokenizing and then performing either tf-idf or word embedding processing on the tokens.

In [23]:
```
from sklearn.model_selection import train_test_split
```

In [24]:
```
yrat = trd.rating
trdd = trd.drop('rating', axis = 1)
```

In [25]:

```
rev = trd.review
```

In [26]:

```
#from nltk import word_tokenize
from nltk.tokenize import RegexpTokenizer

basic_token_pattern = r"(?u)\b\w\w+\b"

tokenizer = RegexpTokenizer(basic_token_pattern)

```

In [ ]:

```

```

In [27]:

```
from nltk import FreqDist
```

First tokenizing, then creating the distribution frequencies for all invidual items.

In [28]:

```
def rowFrs(reviews):
    Frds= []

    for i in trd.review:
        lw= i.lower()
        tkn = tokenizer.tokenize(lw)
        Frd= FreqDist(tkn)

        Frds.append(Frd)
    return Frds
```

In [29]:

```
Frqs= rowFrs(rev)
```

In [30]:

```
len(Frqs)
```

Out[30]: 160398

Expand below to include all rows, i.e. creating the frequency distribution for the corpus. Also choose a sub-sample space for this project of 16,000 (10% of total) for better computing.

In [31]:

```
com = trd[0:16000]
```

In [32]:

```
comb = ""
for i in com['review']:
    comb += str(i)
```

In [33]:

```
comb = comb.lower()
```

In [34]:

```
ttestw= tokenizer.tokenize(comb)
```

In [35]:
```python
from nltk.corpus import stopwords

```

In [36]:
```python

stopwords_list = stopwords.words('english')

w_words_stopped = [word for word in ttestw if word not in stopwords_lis
```

In [37]:
```python
w_words_stoppedC = [word for word in w_words_stopped if '039' not in wo
```

In [38]:
```python
FD= FreqDist(w_words_stoppedC)
```

Converting to dataframe to sort and get top 200 words.

In [39]:
```python
df = pd.DataFrame(data = dict(FD), index = range(16000))
```

In [40]:
```python
dft=df.transpose()
```

In [41]:
```python
dft.head()
```

Out[41]:

	0	1	2	3	4	5	6	7	8	9	...	15990	15991
side	5259	5259	5259	5259	5259	5259	5259	5259	5259	5259	...	5259	5259
effect	1066	1066	1066	1066	1066	1066	1066	1066	1066	1066	...	1066	1066
take	4217	4217	4217	4217	4217	4217	4217	4217	4217	4217	...	4217	4217
combination	145	145	145	145	145	145	145	145	145	145	...	145	145
bystolic	22	22	22	22	22	22	22	22	22	22	...	22	22

5 rows × 16000 columns

In [42]:

```
dft.sort_values(by= 0, axis = 0, ascending = False)
```

Out[42]:

	0	1	2	3	4	5	6	7	8	9	...	15990	159
day	5810	5810	5810	5810	5810	5810	5810	5810	5810	5810	...	5810	58
side	5259	5259	5259	5259	5259	5259	5259	5259	5259	5259	...	5259	52
taking	5085	5085	5085	5085	5085	5085	5085	5085	5085	5085	...	5085	50
years	4649	4649	4649	4649	4649	4649	4649	4649	4649	4649	...	4649	46
pain	4519	4519	4519	4519	4519	4519	4519	4519	4519	4519	...	4519	45
...	...	...	...	...	...	...	...	...	...	...	...	...	
lobular	1	1	1	1	1	1	1	1	1	1	...	1	
peritoneum	1	1	1	1	1	1	1	1	1	1	...	1	
acknowledged	1	1	1	1	1	1	1	1	1	1	...	1	
ai	1	1	1	1	1	1	1	1	1	1	...	1	
inflammed	1	1	1	1	1	1	1	1	1	1	...	1	

20209 rows × 16000 columns

In [43]:

```
df200 =dft.sort_values(by= 0, axis = 0).tail(200)
```

Adic is the inverse frequency part of tf-idf.

In [44]:

```
adic={}
for i in range(len(df200[0])):
    indx =df200.index[i]
    adic[indx]=df200[0][i]


```

Data modelling

Baseline tf-idf model.

Creating the tf-idf matrix.

In [45]:
```python

wtfids = []
for i in range(16000):
    tfidfs =list([])
    for k in adic.keys():
        num = Frqs[i].get(k,0)
        den = adic[k]
        tfidf = num/den
        tfidfs.append(tfidf)
    wtfids.append(tfidfs)
```

Taking the first 16000 of the y-var (rating).

In [46]:
```python
revscut= trd.rating[0:16000]
```

In [47]:
```python
yvals = np.asarray(revscut)
```

In [48]:
```python
xvals = np.asarray(wtfids)
```

In [49]:
```python
x_train, x_val, y_train,  y_val = train_test_split(xvals, yvals, randon
```

Running a baseline linear regression model:

In [50]:
```python
from sklearn import linear_model
reg = linear_model.LinearRegression()




```

In [51]:
```python
reg.fit(x_train, y_train)
```

Out[51]: LinearRegression()

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [52]:
```python
preds = reg.predict(x_train)
```

In [53]:
```python
from sklearn.metrics import mean_squared_error
```

In [54]:
```python
mean_squared_error(y_train, preds)
```

Out[54]: 8.141032346511635

In [55]:
```
rmse = mean_squared_error(y_train, preds)**.5
rmse
```
Out[55]: 2.853249436434122

In [56]:
```
valpreds = reg.predict(x_val)
```

In [57]:
```
mean_squared_error(y_val, valpreds)
```
Out[57]: 8.12230730226126

In [58]:
```
rmse = mean_squared_error(y_val, valpreds)**.5
rmse
```
Out[58]: 2.8499661931786595

Not a lot, if any, overfitting (8.09 mse training vs 8.28 mse validation)

In [59]:
```
from sklearn.tree import DecisionTreeRegressor
```

Baseline mse is 8, scale is 10 points.

Try regression tree.

In [60]:
```
regr1 = DecisionTreeRegressor(max_depth = 3)

```

In [61]:
```
regr1.fit(x_train, y_train)
```
Out[61]: DecisionTreeRegressor(max_depth=3)

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [62]:
```
prds4 = regr1.predict(x_train)
```

In [63]:
```
mean_squared_error(y_train, prds4)
```
Out[63]: 10.176464790716771

In [64]:
```
rmse = mean_squared_error(y_train, prds4)**.5
rmse
```
Out[64]: 3.190057176715924

```
In [65]:  regr1.feature_importances_
```

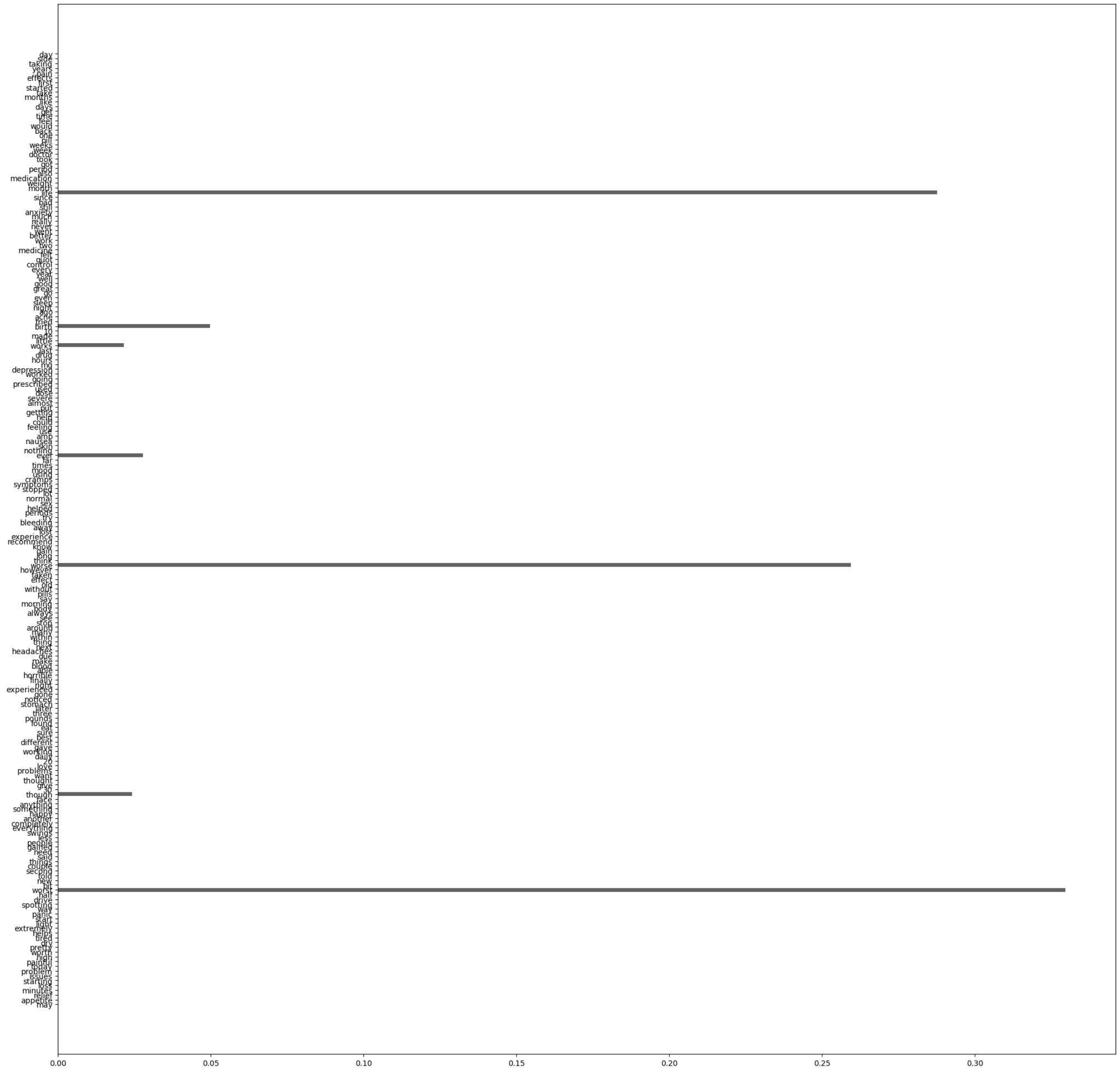
```
Out[65]: array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.32960934,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.02415096,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.25952593, 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.02778917, 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.02157691, 0.        ,
       0.        , 0.        , 0.04972054, 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.28762715, 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

In [66]:

```python
#fig, ax =fig
plt.figure(figsize=(25,25))
plt.barh(list(adic.keys()), regr1.feature_importances_)
```

Out[66]:

```
<BarContainer object of 200 artists>
```



Most important features: worse, love

In [ ]:

```python

```

In [67]:

```python
prds4val = regr1.predict(x_val)
```

In [68]:

```python
mean_squared_error(y_val, prds4val)
```

Out[68]:

```
9.898097270874588
```

In [69]:

```python
rmse = mean_squared_error(y_val, prds4val)**.5
rmse
```

Out[69]:

```
3.1461241664744555
```

Again, not much overfitting. Try tree with greater depth to reduce underfitting:

In [111]:
```
regr1a = DecisionTreeRegressor(max_depth = 5)

```

In [112]:
```
regr1a.fit(x_train, y_train)
```

Out[112]: DecisionTreeRegressor(max_depth=5)

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [113]:
```
prds4 = regr1a.predict(x_train)
```

In [114]:
```
mean_squared_error(y_train, prds4)
```

Out[114]: 9.693739045289481

In [115]:
```
rmse = mean_squared_error(y_train, prds4)**.5
rmse
```

Out[115]: 3.1134770025310097

In [116]:
```
prds4val = regr1a.predict(x_val)
```

In [117]:
```
mean_squared_error(y_val, prds4val)
```

Out[117]: 9.58856602641099

In [118]:
```
rmse = mean_squared_error(y_val, prds4val)**.5
rmse
```

Out[118]: 3.0965409776734734

Allowing for additional depth, reduced underfitting, slightly.

In [70]:
```
from sklearn.ensemble import RandomForestRegressor
```

In [71]:
```
rfr = RandomForestRegressor(max_samples = 100)

```

In [72]:
```
rfr.fit(x_train, y_train)
```

Out[72]: RandomForestRegressor(max_samples=100)

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [73]:

```
prds5 = rfr.predict(x_train)
```

In [74]:

```
mean_squared_error(y_train, prds5)

```

Out[74]: 10.041323552083334

In [75]:

```
rmse = mean_squared_error(y_train, prds5)**.5
rmse
```

Out[75]: 3.16880475133501

In [76]:

```
1 rfr.feature_importances_
```

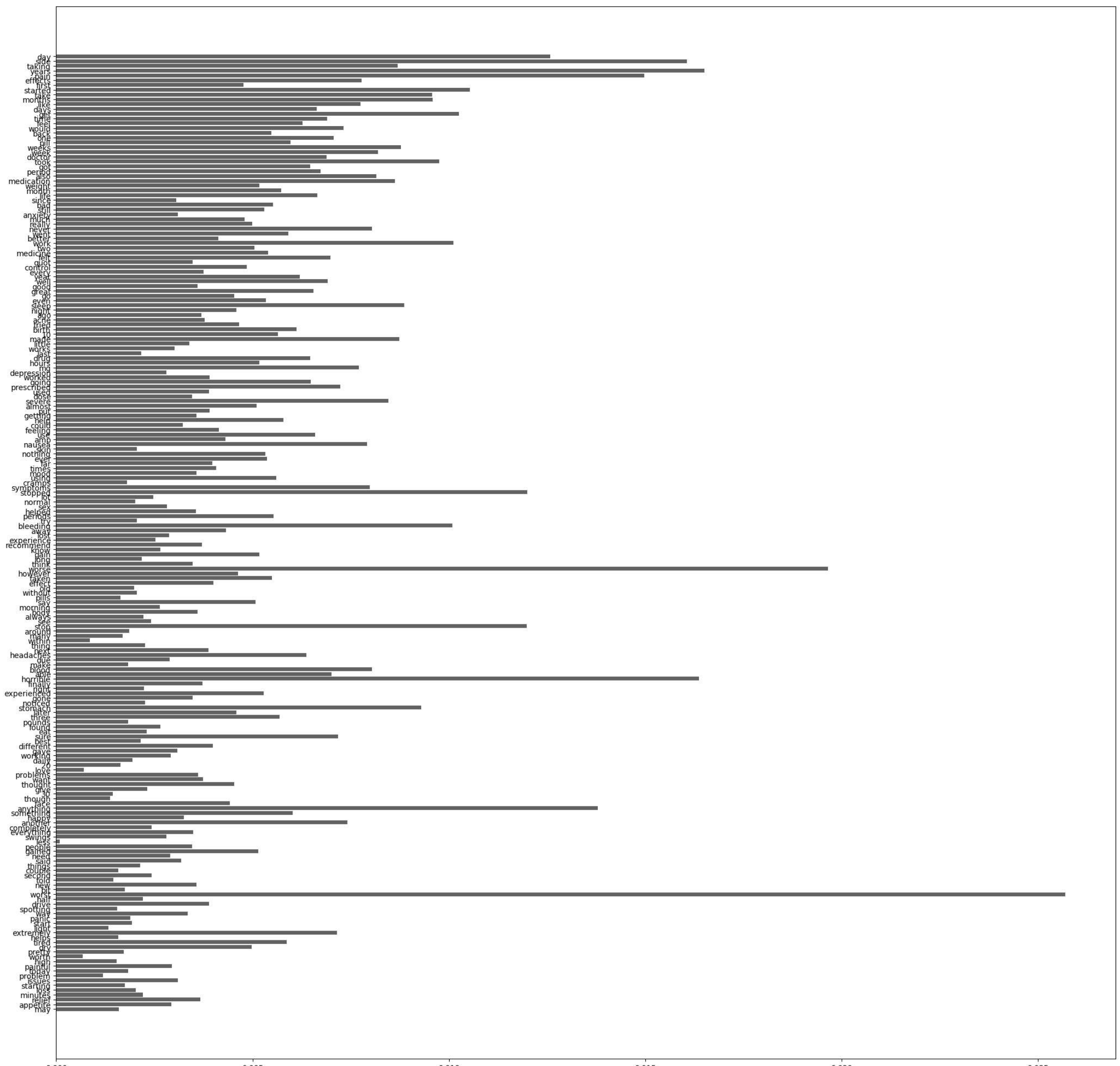
Out[76]:

```
array([1.59724925e-03, 2.92882676e-03, 3.66769522e-03, 2.20346148e-03,
       2.02605297e-03, 1.74539861e-03, 3.09409270e-03, 1.19580705e-03,
       1.83151080e-03, 2.94923648e-03, 1.54433986e-03, 6.82686009e-04,
       1.71982078e-03, 4.97258893e-03, 5.87008048e-03, 1.57862105e-03,
       7.14692117e-03, 1.32418918e-03, 1.92824182e-03, 1.88379885e-03,
       3.34520885e-03, 1.54996728e-03, 3.89108576e-03, 2.20496470e-03,
       2.56976766e-02, 1.74484828e-03, 3.56965373e-03, 1.46243933e-03,
       2.43153449e-03, 1.58514255e-03, 2.14382767e-03, 3.18572508e-03,
       2.90347024e-03, 5.14295401e-03, 3.46801231e-03, 9.10728618e-05,
       2.80920417e-03, 3.48969029e-03, 2.43715906e-03, 7.42320202e-03,
       3.24789272e-03, 6.01912614e-03, 1.38003102e-02, 4.42630889e-03,
       1.37410057e-03, 1.44228476e-03, 2.32370287e-03, 4.53304420e-03,
       3.73370609e-03, 3.62058961e-03, 7.02993025e-04, 1.64236768e-03,
       1.94799204e-03, 2.91253067e-03, 3.08350402e-03, 3.99440248e-03,
       2.15857357e-03, 7.17630671e-03, 2.30590118e-03, 2.64930913e-03,
       1.82635125e-03, 5.68673091e-03, 4.59311607e-03, 9.29668907e-03,
       2.26022060e-03, 3.47791806e-03, 5.28307460e-03, 2.23215751e-03,
       3.72579555e-03, 1.63655943e-02, 7.00746820e-03, 8.03857770e-03,
       1.83792067e-03, 2.88618948e-03, 6.36872223e-03, 3.88106130e-03,
       2.25729499e-03, 8.58336825e-04, 1.69052114e-03, 1.86526550e-03,
       1.19852927e-02, 2.41385803e-03, 2.21799980e-03, 3.59467243e-03,
       2.63471642e-03, 5.07351874e-03, 1.63698361e-03, 2.04837139e-03,
       1.98417887e-03, 4.00631978e-03, 5.49955591e-03, 4.63285850e-03,
       1.96565342e-02, 3.47599240e-03, 2.18154223e-03, 5.16858937e-03,
       2.65239222e-03, 3.71009647e-03, 2.53121020e-03, 2.87405170e-03,
       4.32670130e-03, 1.00923567e-02, 2.05207923e-03, 5.53055891e-03,
       3.56276079e-03, 2.81558922e-03, 2.01353052e-03, 2.47223789e-03,
       1.19995591e-02, 7.98734479e-03, 1.80059698e-03, 5.60059851e-03,
       3.57633134e-03, 4.07868647e-03, 3.98085879e-03, 5.36365373e-03,
       5.33341151e-03, 2.05645301e-03, 7.91505495e-03, 4.30614623e-03,
       6.59837443e-03, 4.14059762e-03, 3.21931024e-03, 5.78170191e-03,
       3.57819575e-03, 3.90089549e-03, 5.10759231e-03, 8.46118962e-03,
       3.45617028e-03, 3.89237695e-03, 7.23313153e-03, 6.48212089e-03,
       3.91090714e-03, 2.81331786e-03, 7.70790814e-03, 5.17183539e-03,
       6.47056522e-03, 2.16921148e-03, 3.01411610e-03, 3.39802068e-03,
       8.73544727e-03, 5.64530677e-03, 6.11569629e-03, 4.65566007e-03,
       3.77959148e-03, 3.69593299e-03, 4.58681763e-03, 8.86117812e-03,
       5.34312346e-03, 4.52904753e-03, 6.55327904e-03, 3.59485704e-03,
       6.90919864e-03, 6.20740454e-03, 3.76001433e-03, 4.85225619e-03,
       3.48133307e-03, 6.97817021e-03, 5.39979101e-03, 5.05397024e-03,
       1.01128959e-02, 4.12749498e-03, 5.91866437e-03, 8.04313659e-03,
       4.99464056e-03, 4.80191214e-03, 3.09557764e-03, 5.30371202e-03,
       5.52319929e-03, 3.06128680e-03, 6.65671795e-03, 5.73483427e-03,
       5.17309312e-03, 8.62691359e-03, 8.15454294e-03, 6.73667672e-03,
       6.47400521e-03, 9.75125816e-03, 6.88339351e-03, 8.19859543e-03,
       8.77826592e-03, 5.97054216e-03, 7.07217512e-03, 5.47969119e-03,
       7.32169771e-03, 6.26942090e-03, 6.90624665e-03, 1.02519291e-02,
       6.64251997e-03, 7.74829323e-03, 9.58478947e-03, 9.58174611e-03,
       1.05318858e-02, 4.77700541e-03, 7.78419558e-03, 1.49822595e-02,
       1.65055295e-02, 8.70090597e-03, 1.60698130e-02, 1.25782653e-02])
```

In [77]:

```
plt.figure(figsize=(25,25))
plt.barh(list(adic.keys()), rfr.feature_importances_)
```

Out[77]: <BarContainer object of 200 artists>



Important features: horrible, worst, doctor

In [78]:

```
prds5val = rfr.predict(x_val)
```

In [79]:

```
mean_squared_error(y_val, prds5val)

```

Out[79]: 9.995630406250003

In [80]:

```
rmse = mean_squared_error(y_val, prds5val)**.5
rmse
```

Out[80]: 3.161586691243813

Baseline continues to perform better.

word embeddings, premade

Use word-embeddings to process input data so that some word-meaning is caputured. Use Stanford's premade word-embedding, "Glove". The W2vectorizer object has method, transorm, which creates an array that has the glove vector for words in the glove dictionary, and rows of 0's for words not in the glove dictionary.

In [81]:
```
total_vocabulary = set(word for word in w_words_stoppedC)
```

In [82]:
```
len(total_vocabulary)
```

Out[82]: 20209

In [83]:
```
glove = {}
with open('glove.6B.50d.txt', 'rb') as f:
    for line in f:
        parts = line.split()
        word = parts[0].decode('utf-8')
        if word in total_vocabulary:
            vector = np.array(parts[1:], dtype=np.float32)
            glove[word] = vector
```

In [84]:
```
class W2vVectorizer(object):

    def __init__(self, w2v):
        # Takes in a dictionary of words and vectors as input
        self.w2v = w2v
        if len(w2v) == 0:
            self.dimensions = 0
        else:
            self.dimensions = len(w2v[next(iter(glove))])

    def fit(self, X, y):
        return self
    # Gets the mean vector from all different words in the particular r
    def transform(self, X):
        return np.array([
            np.mean([self.w2v[w] for w in words if w in self.w2v]
                   or [np.zeros(self.dimensions)], axis=0) for words ir
```

In [85]:
```
t1 = W2vVectorizer(glove)
```

In [86]:
```
t1.dimensions
```

Out[86]: 50

In [87]:
```
t2 = t1.transform(Frqs)
```

In [88]:
```
np.shape(t2)
```
Out[88]: (160398, 50)

In [89]:
```
ys=trd.rating[0:16000]
t2 =t2[0:16000]
```

In [90]:
```
x_trainW, x_valW, y_trainW,  y_valW = train_test_split(t2, ys, random_s
```

In [ ]:
```

```

In [91]:
```
regW = linear_model.LinearRegression()




```

In [92]:
```
regW.fit(x_trainW, y_trainW)
```
Out[92]: LinearRegression()

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [93]:
```
predsW = regW.predict(x_trainW)
```

In [94]:
```
mean_squared_error(y_trainW, predsW)
```
Out[94]: 9.3175497555418

In [95]:
```
rmse = mean_squared_error(y_trainW, predsW)**.5
rmse
```
Out[95]: 3.0524661759865253

In [97]:
```
predsW = regW.predict(x_valW)
```

In [98]:
```
mean_squared_error(y_valW, predsW)
```
Out[98]: 9.242511595713813

In [100]:
```
rmse = mean_squared_error(y_valW, predsW)**.5
rmse
```
Out[100]: 3.040149929808366

Baseline model still performing better.

In [101]:
```
from sklearn.tree import DecisionTreeRegressor
```

In [102]:
```
regr1w = DecisionTreeRegressor(max_depth = 3)

```

In [103]:
```
regr1w.fit(x_trainW, y_trainW)
```

Out[103]:
```
DecisionTreeRegressor(max_depth=3)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [104]:
```
prds3w = regr1w.predict(x_trainW)
```

In [105]:
```
mean_squared_error(y_trainW, prds3w)
```

Out[105]:
```
10.059328144618801
```

In [106]:
```
rmse = mean_squared_error(y_trainW, prds3w)**.5
rmse
```

Out[106]:
```
3.171644391261227
```

In [108]:
```
prdsvalW = regr1w.predict(x_valW)
```

In [109]:
```
mean_squared_error(y_valW, prdsvalW)
```

Out[109]:
```
9.963422304715259
```

In [110]:
```
rmse = mean_squared_error(y_valW, prdsvalW)**.5
rmse
```

Out[110]:
```
3.156488920416997
```

Baseline model still performing better.

Results/conclusions

rmses

- Deployment of baseline model for ‘rating extraction’ from written review.
- Gather insights on how patients rate drugs
    - "doctor, love, worse" etc.

Repository strucuture

- Notebook
- README
- data

In [ ]:

```
1
```