Text Based Suicide Risk Detection Model for Mental Health Chatbots

Author: Roshni Janakiraman

Notebook #1 : Introduction and Data Preparation

I. Business Case

Business Objective & Stakeholders

To provide a suicide risk detection model for mental health chatbot companies (e.g. Pyx, WoeBot

Business Understanding

- There is an increasing need for accessible and affordable mental health care services, given the rising rates of mental illness and the growing shortage of mental health professionals in the United States over recent years.
- The growing Mental Health AI market has played a crucial role in filling this gap, by providing intelligent chatbots that can provide mental health support on-demand.
- Over 40% of Americans exclusively use chatbot services over in-person therapy, and most report satisfaction with these services.
- However, mental health chatbots are limited in their abilities.
- There are populations for whom mental health chatbots are not yet able to provide suitable care, such as **clients at risk for suicide.**
- Unconstrained chatbots have been shown to ignore and even encourage self-harm and suicide.

Problem

- It is essential for mental health chatbots to be able to detect suicide risk and respond appropriately, given 5% of the US population reported experiencing suicidal thoughts
  1. Serious legal and ethical ramifications of failing to respond properly to suicide risk
  2. This will allow mental health companies to better implement their safeguards for high-risk clients
  3. A model that can detect suicide risk will **increase the efficiency of care** of mental health chatbots

Project Goals:

To create a model to classify individuals as **at suicide risk** or **not at suicide risk** based on a text analysis of their messages, using the following features: *word relevance*, *sentiment analysis*, and *emotion detection*

1. High Accuracy Rate

- Model should accurately classify text as indicative of suicide risk vs. non risk

2. High Recall Rate: Minimize *False Negatives*

- False Negative are failing to detect suicide risk in clients who *actually are* at risk
- False negative classifications are *dangerous*, since clients who need intervention would not get the support they need.

3. Quick Run Time

- Model should generate predictions from unseen text *efficiently*, so that it can be deployed to work *in real time* with the mental health chatbot

---

II. Notebook Set-Up

In [1]:
```
'''Python Standard Packages'''
import json
import pickle
import re
import time

'''Anaconda standard packages'''
import matplotlib.pyplot as plt
from nltk.tokenize import TweetTokenizer
from nltk.corpus import stopwords
import numpy as np
import pandas as pd
import seaborn as sns

'''Third Party Packages'''
import contractions
from lingua import Language, LanguageDetectorBuilder
from ftlangdetect import detect
from nrclex import NRCLex
import regex
from spellchecker import SpellChecker
from textblob import TextBlob
from textblob import Word
from textacy import preprocessing as preproc
from tqdm import tqdm
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
```

In [2]:
```
%precision %.3f
pd.set_option('display.float_format', '{:.2f}'.format)

tqdm.pandas()
```

III. Data Understanding

Data Source

Suicide and Depression Dataset from Kaggle

- This dataset consists over **over 200,000** posts webscraped from Reddit between December 2008 - January 2021 using Pushshift API. Posts were collected from "r/SuicideWatch" and "r/Teenagers"

Target: Suicide Risk Classification ("Class")

- **Suicide Risk**: text from "r/SuicideWatch," a forum that provides "peer support for anyone struggling with suicidal thoughts."

- **No Suicide Risk**: text from "r/Teenagers," a forum for "average teenager discussions"

Dataset - Descriptives

- Dataset consists of only two columns: *raw, unclean* text and suicide risk classification (suicide vs. non-suicide)
- Original dataset is too large for GitHub -- can be downloaded here

In [3]:
```
#  Loading in dataset and dropping unnecessary index column
df = pd.read_csv("../data/Suicide_Detection.csv").drop('Unnamed: 0', axis=1)

#  Viewing first 5 rows of df
print(len(df))
df.head()
```

```
232074
```

Out[3]:

	text	class
0	Ex Wife Threatening SuicideRecently I left my ...	suicide
1	Am I weird I don't get affected by compliments...	non-suicide
2	Finally 2020 is almost over... So I can never ...	non-suicide
3	i need helpjust help me im crying so hard	suicide
4	I'm so lostHello, my name is Adam (16) and I'v...	suicide

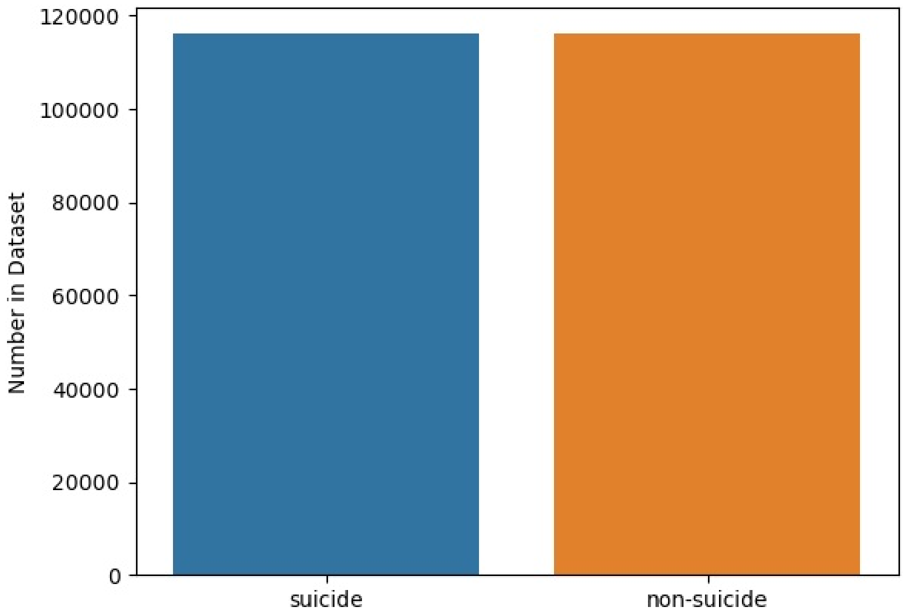
Class Descriptives

- 50/50 split between each class in dataset
- This is notable because class imbalances present in most suicide-related datasets

In [4]:
```
df['class'].value_counts()
```

Out[4]:
```
suicide        116037
non-suicide    116037
Name: class, dtype: int64
```

In [5]:
```
sns.barplot(x=df['class'].value_counts().index, y=df['class'].value_counts().values).set_ylabel('Number in Data:

# plt.savefig('./images/2-classdist.png', bbox_inches='tight', pad_inches=0.1, facecolor='white', transparent=F
```



Class Examples

In [6]:
```
print(f"Example of a suicide-risk post: \n\n {df['text'].loc[0]}")
print("\n")
print(f"Example of a non suicide-risk post: \n\n {df['text'].loc[1]}")
```

```
Example of a suicide-risk post:

 Ex Wife Threatening SuicideRecently I left my wife for good because she has cheated on me twice and lied to me
so much that I have decided to refuse to go back to her. As of a few days ago, she began threatening suicide. I
have tirelessly spent these paat few days talking her out of it and she keeps hesitating because she wants to b
elieve I'll come back. I know a lot of people will threaten this in order to get their way, but what happens if
she really does? What do I do and how am I supposed to handle her death on my hands? I still love my wife but I
cannot deal with getting cheated on again and constantly feeling insecure. I'm worried today may be the day she
does it and I hope so much it doesn't happen.


Example of a non suicide-risk post:

 Am I weird I don't get affected by compliments if it's coming from someone I know irl but I feel really good w
hen internet strangers do it
```

Dataset - Features of Interest

Features of Interest that I will derive from text:

1. **Relevant Keywords:** determined by Term Frequency-Inverse Document Frequency calculation
2. **Sentiment Analysis:** rating the positivity, neutrality & negativity of sentences using *VADER Sentiment*

3. **Emotion Detection:** determining emotions indicated in text by calculating frequency of words associated with eight primary emotions using *NRCLex*

Data Quality

Notable strengths about current dataset:

1. **Large Sample Size** - hard to find mental health data of this size, especially with confidentiality constraints
2. **Equal Representation of Suicide Risk vs. Non-Risk class**

Notable weaknesses:

1. Raw text needs *a lot* of cleaning -- lots of misspellings, unidentifiable characters, inconsistent spacing
   - Even with intensive cleaning techniques, I could not completely clean the data.
   - Trade-off between clean data and keeping authenticity of the data: for example, running a spellcheck altered the meaning of many sentences, so was ultimately not feasible.
2. Control group being r/teenagers -- in future studies, would be better to use a group more indicative of mental health support seekers
3. Spam Posts
   - Many spam posts in the dataset are formatted well enough to avoid spam filters, making it difficult to detect & delete them

---

IV. Data Preparation

A. Text Cleaning Steps

1. Simple Spam Remover

In [7]:
```
def char_counter(string):
    charCount = {"alnum": 0, "not_alnum": 0}
    accepted = " '"
    for char in string:
        if char.isalnum() or char in accepted:
            charCount["alnum"] += 1
        else:
            charCount["not_alnum"] += 1
    if charCount["not_alnum"] > charCount["alnum"]:
        return 0
    else:
        return string
```

In [8]:
```
before = len(df)

df["text_trial"] = df["text"].apply(lambda x: char_counter(x))
df = df[df["text_trial"] != 0]
df = df.drop(["text_trial"], axis=1)

after = len(df)

print(f"CharCounter Removed {before - after} spam data points")
```

```
CharCounter Removed 767 spam data points
```

In [9]:
```
df['text'].isna()
```

Out[9]:
```
0         False
1         False
2         False
3         False
4         False
          ...
232069    False
232070    False
232071    False
232072    False
232073    False
Name: text, Length: 231307, dtype: bool
```

2. Removing HTML tags, usernames (u/name), website urls, and numbers

In [10]:
```
""" TweetTokenizer will isolate HTML characters, easier to remove """
```

```python
twtokenizer = TweetTokenizer()
df['text'] = df['text'].apply(lambda x: " ".join(twtokenizer.tokenize(x)))
```

In [11]:
```python
def special_char_remover(row):
    patterns = r'|'.join(map(r'(?:{})'.format,
                             (r"\n&\S+", r"\n", r"&lt", r"&gt",
                              r"u/\S+", r"ww\S+", r"htt\S+",
                              r"\d\S+", r"\d+")))
    row = re.sub(patterns, '', row)
    str_en = row.encode("ascii", "ignore")
    str_de = str_en.decode()
    return str_de


df["text"] = df["text"].apply(lambda x: special_char_remover(x))
```

3. Expanding Contractions

In [12]:
```python
df['text'] = df['text'].apply(lambda x: contractions.fix(x))
```

4. Removing Non-English Posts

In [13]:
```python
detector = LanguageDetectorBuilder.from_all_languages().with_preloaded_language_models().build()
```

In [14]:
```python
def quick_detect(row):
    word_dict = detect(text=row)
    lang_tuple = (word_dict['lang'], word_dict['score'])
    return lang_tuple
```

In [15]:
```python
quick_detect("I became paranoid that the school of jellyfish was spying on me.")
```

```
Warning : `load_model` does not return WordVectorModel or SupervisedModel any more, but a `FastText` object whi
ch is very similar.
```

Out[15]:
```
('en', 1.000)
```

In [16]:
```python
quick_detect("I feel so silly right now because I did not even think about them being busy because of the \
Super Bowl. I probably should not have gotten so mad at them.")
```

Out[16]:
```
('en', 0.989)
```

In [17]:
```python
df['lang_tuple'] = df['text'].progress_apply(lambda row: quick_detect(row))
```

```
100%|██████████| 231307/231307 [00:05<00:00, 43213.56it/s]
```

In [18]:
```python
df['ft_lang'] = df['lang_tuple'].apply(lambda x: x[0])
df['ft_conf'] = df['lang_tuple'].apply(lambda x: x[1])
df.drop('lang_tuple', axis=1, inplace=True)
```

In [19]:
```python
noten_df = df[df['ft_lang']!= 'en'].sort_values(by='ft_conf', ascending=False)
```

In [20]:
```python
not_en_index = list(noten_df[noten_df['ft_conf']>0.90].index)
```

In [21]:
```python
# Dropping all values from df and noten_df where != 'en' and ft_conf > 0.90

noten_df.drop(not_en_index, inplace=True)
df.drop(not_en_index, inplace=True)
```

In [22]:
```python
print(f"So far, {231307 - len(df)} non-english rows were dropped")
```

```
So far, 104 non-english rows were dropped
```

In [23]:
```python
# Verifying rest with Lingua Detect:
```

In [24]:
```python
def lingua_detection(row):
    if detector.detect_language_of(row) == Language.ENGLISH:
        return 1
    else:
        return 0
```

In [25]:
```python
noten_df['lingua_is_en'] = noten_df['text'].progress_map(lambda x: lingua_detection(x))
```

```
100%|██████████| 1494/1494 [07:34<00:00,  3.28it/s]
```

In [26]:
```python
not_en_index = list(noten_df[noten_df['lingua_is_en']==0].index)

noten_df.drop(not_en_index, inplace=True)
df.drop(not_en_index, inplace=True)
```

In [27]:
```python
print(f"So far, {231307 - len(df)} non-english rows were dropped")
```
```
So far, 641 non-english rows were dropped
```

In [28]:
```python
low_conf_english_df = df[df['ft_conf']<0.5]
```

In [29]:
```python
lc_en_df=low_conf_english_df[low_conf_english_df['ft_lang']=='en'].copy()
```

In [30]:
```python
lc_en_df['lingua_is_en'] = lc_en_df['text'].progress_map(lambda x: lingua_detection(x))
```
```
100%|██████████| 2886/2886 [14:09<00:00,  3.40it/s]
```

In [31]:
```python
not_en_index = lc_en_df[lc_en_df['lingua_is_en']==0].index

df.drop(not_en_index, inplace=True)
lc_en_df.drop(not_en_index, inplace=True)
```

In [32]:
```python
print(f"{231307 - len(df)} non-english rows were dropped")
```
```
1235 non-english rows were dropped
```

5. Textacy Pre-Processing Pipline

In [33]:
```python
# Timed Output: Takes 60 seconds to run

start = time.time()

pipe = preproc.make_pipeline(preproc.normalize.hyphenated_words,
                             preproc.normalize.quotation_marks,
                             preproc.normalize.unicode,
                             preproc.remove.accents,
                             preproc.remove.brackets,
                             preproc.remove.html_tags,
                             preproc.remove.punctuation,
                             preproc.normalize.whitespace)

df["text"] = df["text"].apply(lambda x: pipe(x))

end = time.time()
print(f'Time Taken: {end-start:.1f} seconds')
```
```
Time Taken: 44.7 seconds
```

6. Caps Processing Tasks + Lowercase All

In [34]:
```python
ect_ther = list((df["text"].apply
                 (lambda x: re.findall(r" ECT ", x)).sort_values
                 (ascending=False)[:72].index))
ect_ther.extend([12935, 181494, 21986, 189017, 49323, 78657, 205774])
for index in ect_ther:
    df['text'][index] = re.sub(r'\Wect\W', r' electroconvulsive therapy ', df['text'][index], flags=re.A | re.I
```
```
/var/folders/vn/0fdnf_cd0bsfss0cmbntv5s40000gn/T/ipykernel_31921/897260948.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#ret
urning-a-view-versus-a-copy
  df['text'][index] = re.sub(r'\Wect\W', r' electroconvulsive therapy ', df['text'][index], flags=re.A | re.I |
re.S)
```

In [35]:
```python
def caps_process(row):
    row = re.sub(r"(?<![A-Z\W])(?=[A-Z])", " ", row)
    row = re.sub(r" ect ", r" etc ", row)
    return row.lower()
```

In [36]:
```python
df['text'] = df['text'].apply(lambda x: caps_process(x))
```

7. Noise Correction

In [37]:
```python
def noise_corr(row):

    #  Prefix Fixing
    row = re.sub(r" (anti) (\S+) ", r" \1\2 ", row)
    row = re.sub(r" (mc) (\S+) ", r" \1\2 ", row)

    #  Removing filler words
    row = re.sub(r"fill\S+", "", row)
    row = re.sub(r"zz\S+", "", row)

    #  Remove duplicated words from row
    row = regex.sub(r'(?<= |^)(\S+)(?: \1){2,}(?= |$)', r'\1 \1', row)
```

```
    #  Remove all repeating characters > 2
    row = re.sub(r'(.)\1{2,}', r'\1', row)

    #  Correct cases of tldr being separated
    row = re.sub(r" tl dr ", r" tldr ", row)

    #  Make sure all chars are alnum
    for word in row.split():
        if word.isalnum():
            continue
        else:
            for char in word:
                if not char.isalnum():
                    word = word.replace(char, '')

    #  Return row with normalized whitespace
    return preproc.normalize.whitespace(row)
```

In [38]:
```
df['text'] = df['text'].progress_apply(lambda x: noise_corr(x))
```

```
100%|██████████| 230072/230072 [00:30<00:00, 7485.65it/s]
```

8. Spelling Correction

- While I could not completely clean all misspelled words, I did change the most common errors
    - Applying a spellchecker to the entire DF took a lot of computational power and led to significant meaning changes, which would be worse for the current analysis than spelling errors.
    - This method ended up being the best compromise.
- In the "Testing SpellChecker" section of scratch_notebook, you can see my full process for finding and changing spelling errors.
- To simplify this notebook, I created imports with my changes & ran them with the functions below:

In [39]:
```
spell = SpellChecker()
with open('../cleaning_dictionaries/single_token_list.json', 'r') as f:
    single_token_list = json.load(f)
with open('../cleaning_dictionaries/correction_dictionary.json', 'r') as file:
    correction_dictionary = json.load(file)
```

In [40]:
```
""" Code to fix typos where i is appended to words """

index_check_wordis = list(df['text'][df['text'].apply(
    lambda row:len(re.findall(r"\b\S+i\b", row))) != 0].index)

for index in index_check_wordis:
    row = df['text'][index]
    for word in re.findall(r"\b\S+i\b", row):
        if spell.known([word]):
            continue

        tester = word[:-1]
        if len(spell.known([tester])) == 1:
            row = re.sub(word, tester + " i", row)
            df['text'][index] = row
        else:
            row = re.sub(word, '', row)
            df['text'][index] = row
```

```
/var/folders/vn/0fdnf_cd0bsfss0cmbntv5s40000gn/T/ipykernel_31921/3917712090.py:15: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#ret
urning-a-view-versus-a-copy
  df['text'][index] = row
/var/folders/vn/0fdnf_cd0bsfss0cmbntv5s40000gn/T/ipykernel_31921/3917712090.py:18: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#ret
urning-a-view-versus-a-copy
  df['text'][index] = row
```

In [41]:
```
def single_token_remover(row_list, token_list):
    """Remove Tokens that are single characters"""

    for token in row_list:
        if token in token_list:
            row_list.remove(token)
            single_token_remover(row_list, single_token_list)
```

```
    return " ".join(row_list)
```

In [42]:
```
df["text"] = df["text"].progress_map(
    lambda x: single_token_remover(x.split(), single_token_list))
```

```
100%|██████████| 230072/230072 [01:25<00:00, 2683.95it/s]
```

In [43]:
```
def replace_words(row, dictionary):
    """ Replacement Dictionary """

    new_row = []
    for word in row.split():
        if word in dictionary.keys():
            new_row.append(dictionary[word])
        else:
            new_row.append(word)
    return (" ".join(new_row))
```

In [44]:
```
df["text"] = df["text"].progress_map(
    lambda x: replace_words(x, correction_dictionary))
```

```
100%|██████████| 230072/230072 [00:03<00:00, 71212.35it/s]
```

9. Drop Unnecessary Columns

In [46]:
```
df.drop(['ft_lang', 'ft_conf'], axis=1, inplace=True)
```

B. Feature Engineering

- As rule-based algorithms, VADER and NRCLex can be run on the full df before the train-test split.
  - This will reduce computational effort during modeling & allows for pre-modeling data exploration

1. Sentiment Analysis with VADERSentiment

- Rates positivity, negativity & neutrality of sentences -- scale of (0 - 1)
- Compound score = sum of positive, negative & neutral scores, normalized across VADER's known lexicon
  - Takes into account word placement in sentence and modifiers (see example below)
    - Many other text analyzers (including NRCLex) do not account for this!

VADERSentiment Example

In [47]:
```
sent = SentimentIntensityAnalyzer()

print(sent.polarity_scores("it was good to see them until they ruined \
everything by insulting me and making me feel horrible."))
```

```
{'neg': 0.367, 'neu': 0.524, 'pos': 0.109, 'compound': -0.7845}
```

Getting Vader Sentiment Scores and Joining to DF

In [48]:
```
def sentiment_analyzer(df):
    """ Analyzes df for sentiment scores and concatenates results """
    dictlist = []
    sia = SentimentIntensityAnalyzer()
    for i in df.index:
        polarity = sia.polarity_scores(df['text'][i])
        dictlist.append({'negative': polarity['neg'],
                         'neutral': polarity['neu'],
                         'positive': polarity['pos'],
                         'comp': polarity['compound']})

    sentdf = pd.DataFrame(dictlist, index=df.index)
    df = df.join(sentdf, rsuffix='sent')
    return df
```

In [49]:
```
start = time.time()
df = sentiment_analyzer(df)
end = time.time()
print(f'Time to run: {(end-start)//60:.0f} min and {(end-start)%60:.0f} seconds')
```

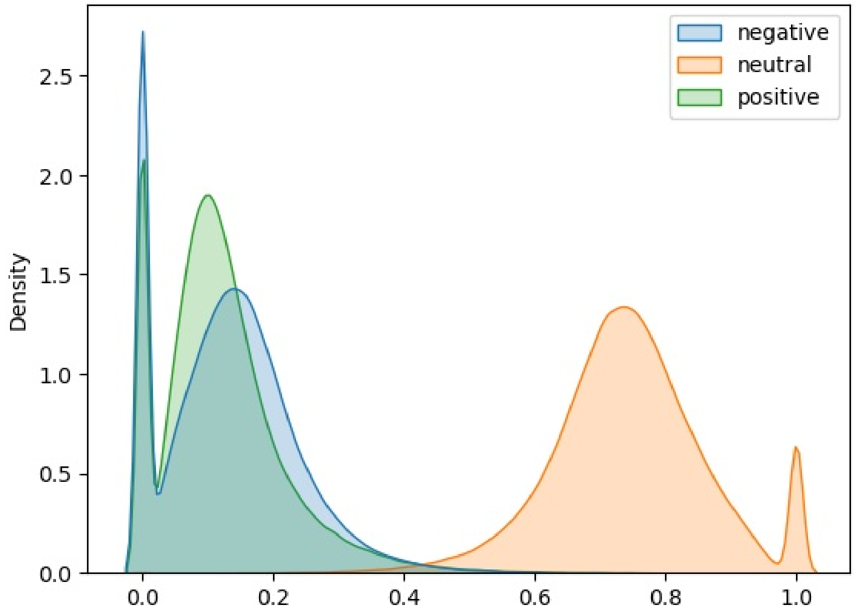
```
Time to run: 5 min and 0 seconds
```

In [50]:
```
df.describe()
```

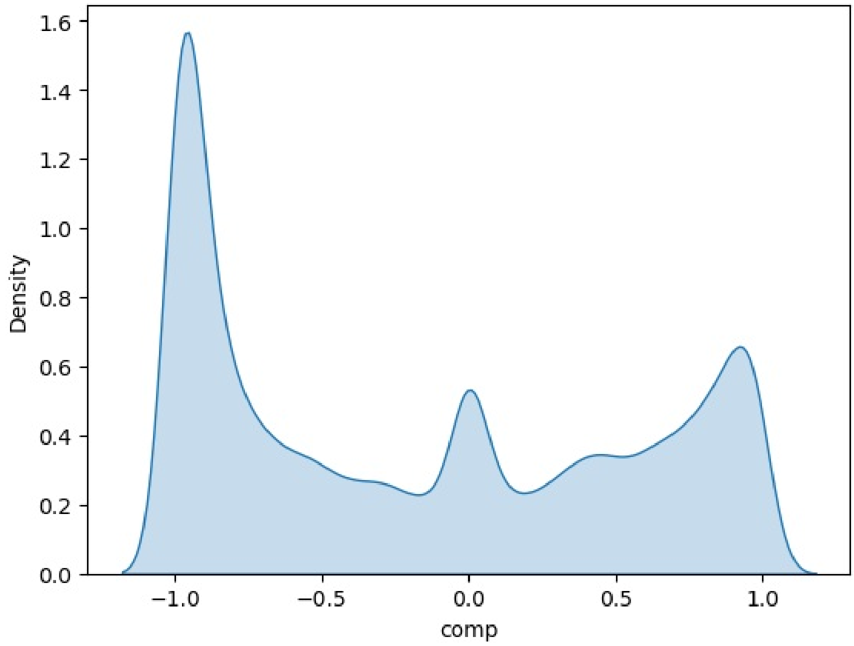
Out[50]:

	negative	neutral	positive	comp
count	230072.00	230072.00	230072.00	230072.00
mean	0.13	0.74	0.12	-0.16
std	0.10	0.12	0.10	0.71
min	0.00	0.00	0.00	-1.00
25%	0.06	0.67	0.06	-0.89
50%	0.13	0.74	0.11	-0.26
75%	0.19	0.81	0.17	0.53
max	1.00	1.00	1.00	1.00

In [51]:
```
#  Distribution of Neg, Neutral & Positive ratings (scale: 0-1)
sns.kdeplot(df.iloc[:,2:5], fill=True);
```



In [52]:
```
sns.kdeplot(df.iloc[:,-1], fill=True);
```



- Needs to be scaled. Data is very skewed.
- Positive & Negative scores skewed right, but neutral scores skewed left.

Not much difference between clean and dirty text in VADER scores, so distribution may not be due to texts being too "sanitized."

In [54]:
```
plotdf = df.drop(['text'], axis=1)
plotdf = pd.DataFrame(plotdf.set_index(['class']).unstack())
plotdf = plotdf.reset_index(level=[0,1])
plotdf.columns=['senttype','class','val']
```

```
plotdf
fig, ax = plt.subplots(figsize=(5,4))
sns.pointplot(plotdf, x='class', y='val', hue='senttype', ax=ax).set_title("Sentiment Means By Class");
# plt.savefig('./images/3-vader.png', bbox_inches='tight', pad_inches=0.1, facecolor='white', transparent=False
```

- Graph indicates large difference in sentiment compound scores between suicide & nonsuicide; rest look insignificant

2. Emotion Detection Analysis with NRCLex

- Based on Plutchik's Wheel of Emotions: counts emotion words of 8 "primary" emotions
- Word coded as emotion based on a large selection of crowdsourced tweets
  - Word-based: not sensitive to context and sentence
    - Built in **tokenizing & lemmatizing** with *TextBlob*
- Returns count or frequency of emotion words used in a sentence
- **I will remove "positive, negative" ratings from NRCLex because VADER is better for Sentiment ratings**
  - Only use ratings of the 8 Emotion Words": Anger, Sadness, Disgust, Fear, Joy, Trust, Anticipation, Surprise

NRCLex Example

In [55]:

```
sentence = NRCLex("it was good to see them until they ruined \
everything by insulting me and making me feel horrible.")


for key, val in sentence.affect_dict.items():
    print(f'{key}: {val}')
print('\n')

pd.DataFrame([sentence.affect_frequencies, sentence.raw_emotion_scores], index=['affect_frequencies', 'raw_emot
```

```
good: ['anticipation', 'joy', 'positive', 'surprise', 'trust']
ruined: ['anger', 'disgust', 'fear', 'negative', 'sadness']
insulting: ['anger', 'disgust', 'fear', 'negative', 'sadness']
horrible: ['anger', 'disgust', 'fear', 'negative']
```

Out[55]:

	fear	anger	anticip	trust	surprise	positive	negative	sadness	disgust	joy	anticipation
affect_frequencies	0.16	0.16	0.00	0.05	0.05	0.05	0.16	0.11	0.16	0.05	0.05
raw_emotion_scores	3.00	3.00	NaN	1.00	1.00	1.00	3.00	2.00	3.00	1.00	1.00

Getting Raw Emotion Scores and Joining to DF

In [56]:

```
emot_map = df['text'].progress_map(lambda x: NRCLex(x).raw_emotion_scores)
```

```
100%|██████████| 230072/230072 [01:11<00:00, 3222.02it/s]
```

In [57]:

```
lex_raw = pd.DataFrame([emot for emot in emot_map.values], index=emot_map.index)
lex_raw = lex_raw.drop(['positive', 'negative'], axis=1)
df = df.join(lex_raw, rsuffix='lex')
df = df.mask(df.isna(), 0)
```

Data Exploration of Emotion Raw Scores

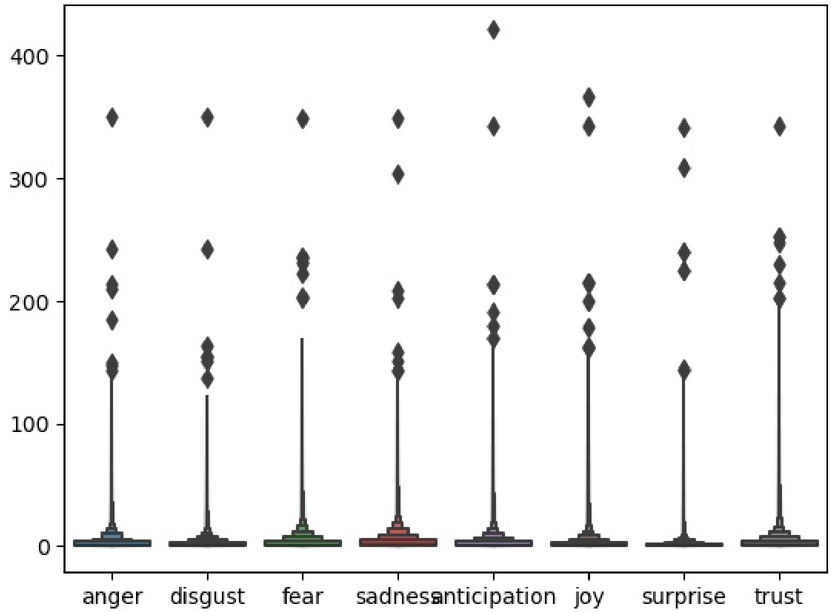
Data Exploration of Emotion Raw Scores

In [58]:
```
df.iloc[:,6:].describe()
```

Out[58]:

	anger	disgust	fear	sadness	anticipation	joy	surprise	trust
count	230072.00	230072.00	230072.00	230072.00	230072.00	230072.00	230072.00	230072.00
mean	2.77	2.13	3.47	3.87	3.10	2.45	1.35	3.44
std	4.81	3.86	5.88	6.37	5.51	4.57	2.94	6.08
min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
25%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
50%	1.00	1.00	1.00	2.00	1.00	1.00	0.00	2.00
75%	4.00	3.00	4.00	5.00	4.00	3.00	2.00	4.00
max	350.00	350.00	349.00	349.00	421.00	367.00	342.00	343.00

In [59]:
```
#  Distribution of Emotions
lex = df.iloc[:,6:]
sns.boxenplot(lex);
```

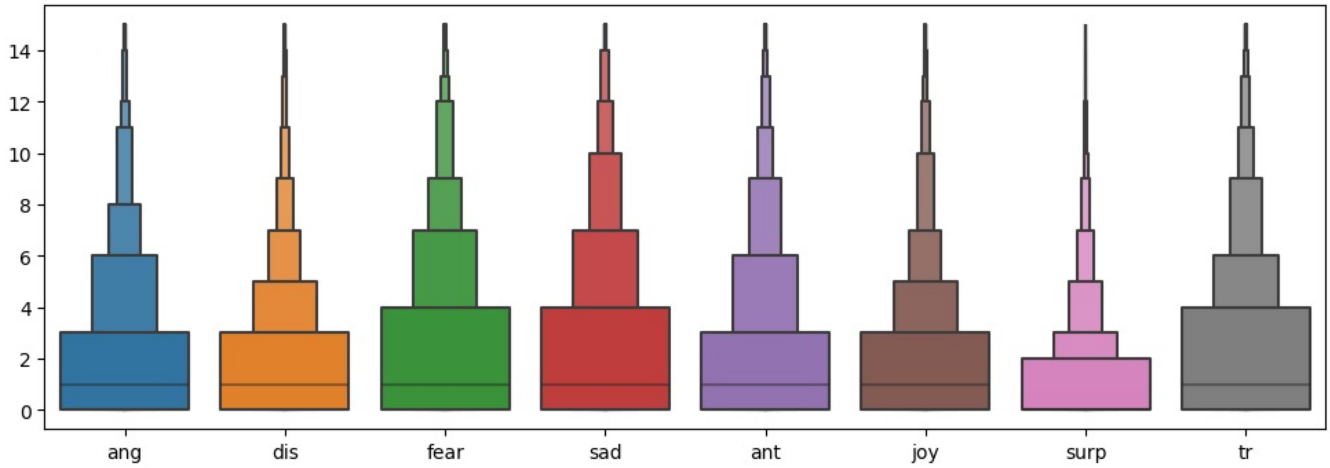


In [60]:
```
lex.quantile(.80)
```

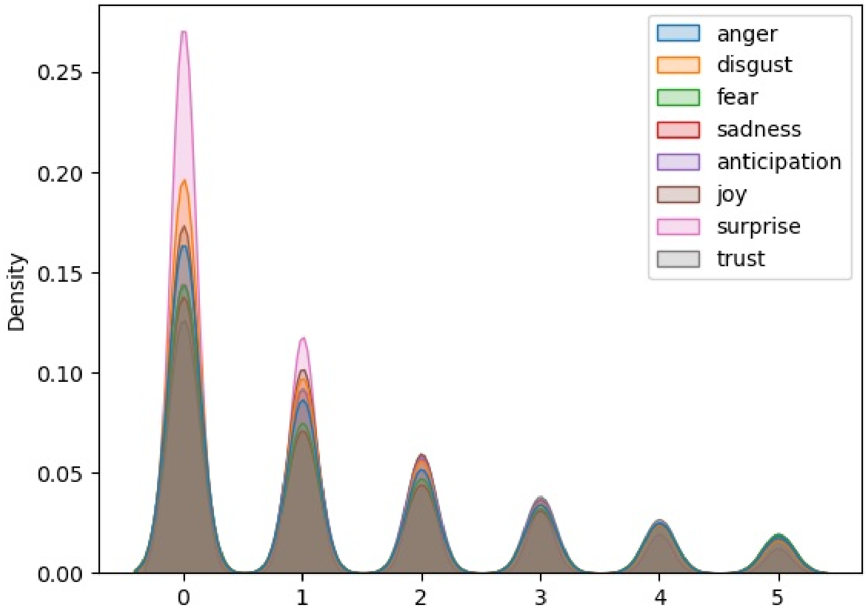
Out[60]:
```
anger           4.00
disgust         3.00
fear            6.00
sadness         6.00
anticipation    5.00
joy             4.00
surprise        2.00
trust           5.00
Name: 0.8, dtype: float64
```

- Heavy outliers! Needs to be scaled, possibly need to remove outliers.
- boxenplot when values capped largest percentile values: 95th (15), 5 (80) :

In [61]:
```
fig, ax = plt.subplots(figsize=(12,4))

sns.boxenplot(lex.mask(lex > 15, np.nan), ax=ax).set_xticklabels(['ang','dis','fear','sad','ant','joy','surp','
```



In [62]:
```
# KDE plot when capped at 80th percentile (x < 5):
sns.kdeplot(lex.mask(lex > 5, np.nan), fill=True);
```



In [63]:
```
df.pivot_table(values=['anger','disgust', 'fear', 'sadness',
            'anticipation', 'joy', 'surprise', 'trust'],
            index='class', aggfunc=np.mean)
```

Out[63]:

class	anger	anticipation	disgust	fear	joy	sadness	surprise	trust
non-suicide	0.96	1.33	0.82	1.03	1.23	1.05	0.65	1.72
suicide	4.57	4.84	3.42	5.88	3.65	6.65	2.05	5.14

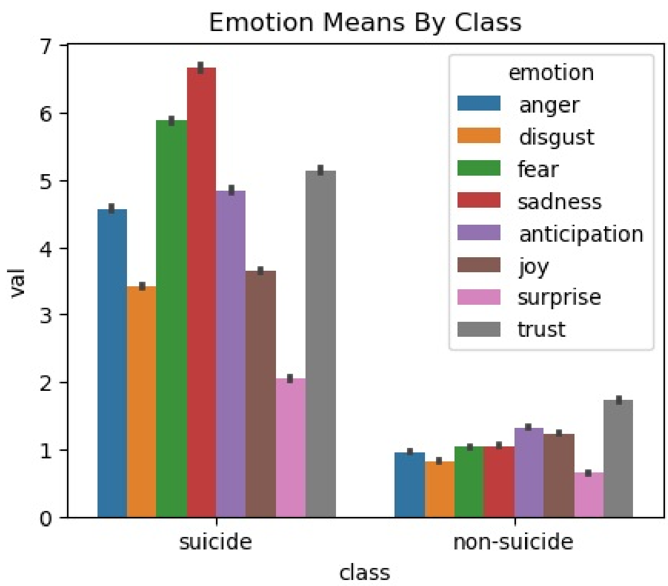
In [64]:
```
df.pivot_table(values=['anger','disgust', 'fear', 'sadness',
            'anticipation', 'joy', 'surprise', 'trust'],
            index='class', aggfunc=max)
```

Out[64]:

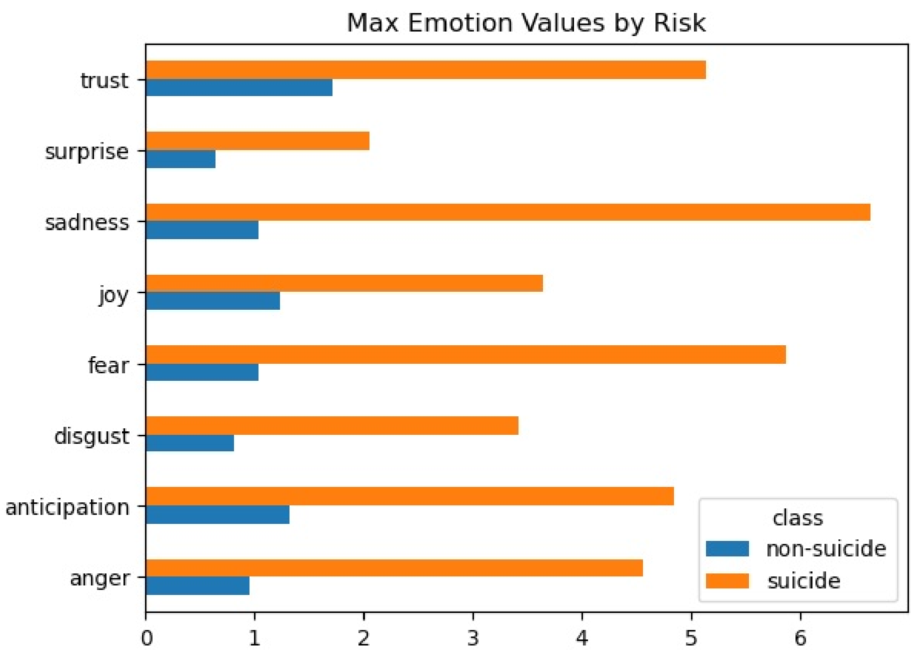
class	anger	anticipation	disgust	fear	joy	sadness	surprise	trust
non-suicide	242.00	421.00	242.00	223.00	367.00	159.00	342.00	343.00
suicide	350.00	191.00	350.00	349.00	179.00	349.00	77.00	248.00

- Mean and Median of *all* emotions greater for suicide risk posts than non-suicide posts
- Max val of positive emotions higher for non-suicide group, but max val for negative emotions higher for suicide group

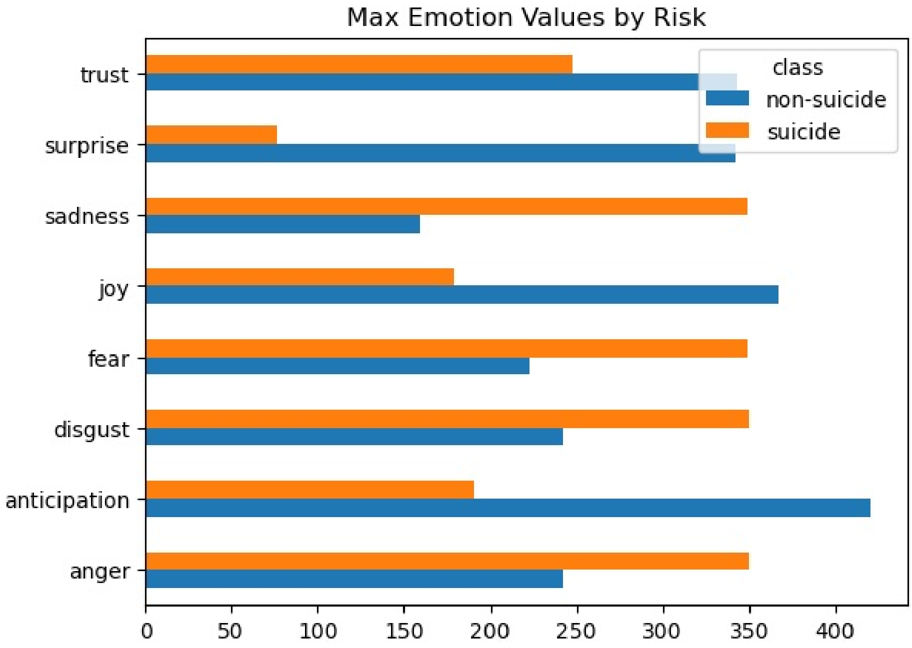
In [65]:
```
eplotdf = df.drop(['text', 'negative', 'neutral', 'positive', 'comp'], axis=1)
eplotdf = pd.DataFrame(eplotdf.set_index(['class']).unstack())
eplotdf = eplotdf.reset_index(level=[0,1])
eplotdf.columns=['emotion','class','val']

fig, ax = plt.subplots(figsize=(5,4))
sns.barplot(eplotdf, x='class', y='val', hue='emotion').set_title("Emotion Means By Class");
```



In [66]:
```
plotdf = df.pivot_table(values=['anger','disgust', 'fear', 'sadness',
            'anticipation', 'joy', 'surprise', 'trust'],
            index='class', aggfunc=np.mean)

plotdf.T.plot(kind='barh').set_title("Max Emotion Values by Risk");
# plt.savefig('./images/2-classdist.png', bbox_inches='tight', pad_inches=0.1, facecolor='white', transparent=Fa
```



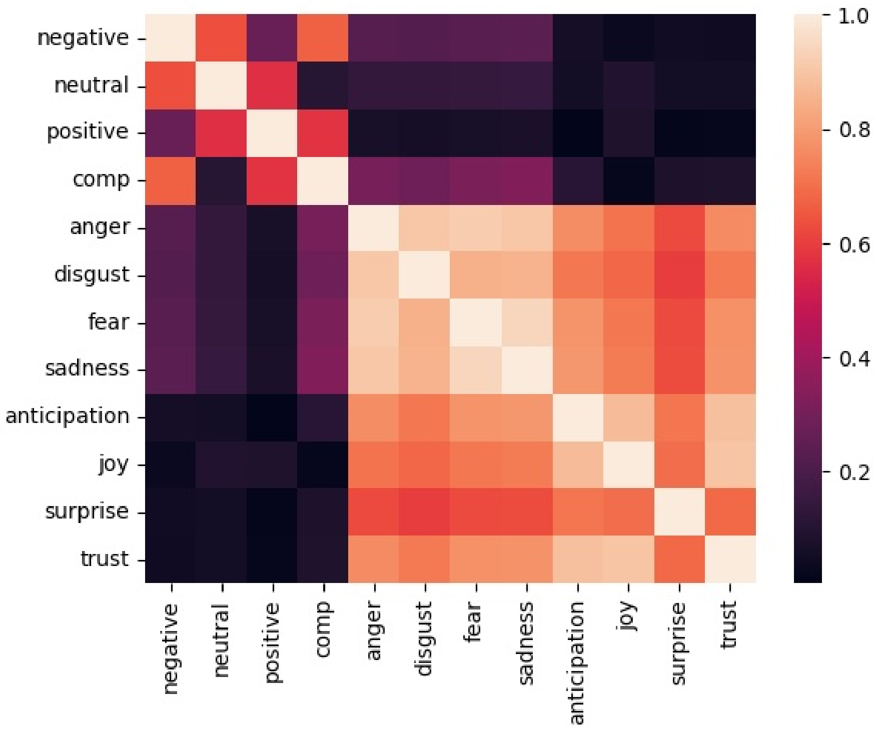
In [67]:
```
plotdf = df.pivot_table(values=['anger','disgust', 'fear', 'sadness',
            'anticipation', 'joy', 'surprise', 'trust'],
            index='class', aggfunc=max)

plotdf.T.plot(kind='barh').set_title("Max Emotion Values by Risk");
# plt.savefig('./images/2-classdist.png', bbox_inches='tight', pad_inches=0.1, facecolor='white', transparent=Fa
```



---

3. Heat Map of Emotion & Sentiment Scores

In [68]:
```
sns.heatmap(abs(df.iloc[:,2:].corr()));
```



- The emotion components seem to be heavily related to one another -- might not represent 8 separate components
- Sentiment components are more distinct from one another, but still correlated
- Low correlation between emotion & sentiment components

Next steps:

- Might be worth doing a PCA analysis with emotion components and reducing into fewer dimensions

---

C. Final Dataframe Preparation

1. Lemmatize + Tokenize text with TextBlob

- Using same Lemmatizer as is in-built in NRCLex (TextBlob)

- Tokenize + rejoin tokens with space since TFIDF has built-in space tokenizer
- Store as df[lem], separate from df[text], in case need to re-run sentiment/emotion analysis

In [69]:
```
## Warning: Take ~12 min to run

start = time.time()

df['tags'] = df['text'].progress_map(lambda x: TextBlob(x).tags)

end = time.time()

print(f"Time to run:{end-start}.")
```

```
100%|██████████| 230072/230072 [11:07<00:00, 344.79it/s]
Time to run:667.2844219207764.
```

In [70]:
```
df['tag_tokens'] = df['tags'].apply(lambda taglist: [(tag[0], tag[1][0].lower()) for tag in taglist])
```

In [71]:
```
acceptedtags = ['a', 'n', 'v', 'r']
```

In [72]:
```
def lemmatize_by_tag_tokens(row):

    new_row = []

    for tag in row:
        if tag[1] in acceptedtags:
            new_row.append(tag[0].lemmatize(tag[1]))
        else:
            if tag[1] == 'j':
                new_row.append(tag[0].lemmatize('a'))
            else:
                new_row.append(tag[0])
    return new_row
```

In [73]:
```
df['lem'] = df['tag_tokens'].apply(lambda x: lemmatize_by_tag_tokens(x))
```

In [74]:
```
df.drop(['tags', 'tag_tokens'], axis=1, inplace=True)
```

2. Remove Stopwords

For TF-IDF analysis, remove stopwords:

In [75]:
```
stop = stopwords.words('english')

df['lem_nostop'] = df['lem'].apply(lambda x: ' '.join([word for word in x if word not in (stop)]))
```

In [79]:
```
df.drop(['text','lem'], axis=1, inplace=True)
```

3. Convert Target to Binary & Format Columns

In [80]:
```
df['target'] = df['class'].replace({'suicide':1, 'non-suicide':0})
df.drop('class',axis=1,inplace=True)
```

4. Rename and Reorganize Columns

In [81]:
```
df2 = df.copy()
```

In [82]:
```
df
```

Out[82]:

	negative	neutral	positive	comp	anger	disgust	fear	sadness	anticipation	joy	surprise	trust	lem_nostop	target
0	0.19	0.73	0.07	-0.95	8.00	4.00	8.00	6.00	7.00	5.00	5.00	5.00	ex wife threaten suicide recently leave wife g...	1
1	0.04	0.76	0.21	0.72	0.00	1.00	0.00	0.00	2.00	1.00	1.00	2.00	weird get affect compliment come someone know ...	0
2	0.26	0.67	0.08	-0.70	2.00	2.00	2.00	1.00	2.00	2.00	1.00	3.00	finally almost never hear bad year ever swear ...	0
3	0.29	0.40	0.31	0.11	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	need help help cry hard	1
4	0.19	0.73	0.08	-1.00	18.00	10.00	27.00	26.00	20.00	6.00	5.00	9.00	lose hello name adam struggle year afraid past...	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
232069	0.08	0.92	0.00	-0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	like rock go get anything go	0
232070	0.09	0.77	0.15	0.34	1.00	1.00	1.00	1.00	0.00	0.00	0.00	1.00	tell many friend lonely everything deprive pre...	0
232071	0.00	0.84	0.16	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	pee probably taste like salty tea someone drin...	0
232072	0.18	0.72	0.10	-0.99	9.00	6.00	7.00	10.00	3.00	1.00	1.00	3.00	usual stuff find post sympathy pity know far b...	1
232073	0.26	0.69	0.05	-0.86	2.00	1.00	3.00	3.00	0.00	0.00	0.00	0.00	still beat first bos hollow knight fight time ...	0

230072 rows × 14 columns

In [ ]:
```
# plt.savefig('./images/2-classdist.png', bbox_inches='tight', pad_inches=0.1, facecolor='white', transparent=Fa
```

In [85]:
```
df.columns
```

Out[85]:
```
Index(['negative', 'neutral', 'positive', 'comp', 'anger', 'disgust', 'fear',
       'sadness', 'anticipation', 'joy', 'surprise', 'trust', 'lem_nostop',
       'target'],
      dtype='object')
```

In [93]:
```
collist = ['neg', 'neu', 'pos', 'compound', 'anger', 'disgust', 'fear',
           'sadness', 'anticipation', 'joy', 'surprise', 'trust',
           'text', 'target']
df.columns = collist
df
```

Out[93]:

	neg	neu	pos	compound	anger	disgust	fear	sadness	anticipation	joy	surprise	trust	text	target
0	0.19	0.73	0.07	-0.95	8.00	4.00	8.00	6.00	7.00	5.00	5.00	5.00	ex wife threaten suicide recently leave wife g...	1
1	0.04	0.76	0.21	0.72	0.00	1.00	0.00	0.00	2.00	1.00	1.00	2.00	weird get affect compliment come someone know ...	0
2	0.26	0.67	0.08	-0.70	2.00	2.00	2.00	1.00	2.00	2.00	1.00	3.00	finally almost never hear bad year ever swear ...	0
3	0.29	0.40	0.31	0.11	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	need help help cry hard	1
4	0.19	0.73	0.08	-1.00	18.00	10.00	27.00	26.00	20.00	6.00	5.00	9.00	lose hello name adam struggle year afraid past...	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
232069	0.08	0.92	0.00	-0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	like rock go get anything go	0
232070	0.09	0.77	0.15	0.34	1.00	1.00	1.00	1.00	0.00	0.00	0.00	1.00	tell many friend lonely everything deprive pre...	0
232071	0.00	0.84	0.16	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	pee probably taste like salty tea someone drin...	0
232072	0.18	0.72	0.10	-0.99	9.00	6.00	7.00	10.00	3.00	1.00	1.00	3.00	usual stuff find post sympathy pity know far b...	1
232073	0.26	0.69	0.05	-0.86	2.00	1.00	3.00	3.00	0.00	0.00	0.00	0.00	still beat first bos hollow knight fight time ...	0

230072 rows × 14 columns

In [94]:
```
orderedcols = ['target', 'text', 'neg', 'neu', 'pos', 'compound',
               'anger', 'disgust', 'fear', 'sadness',
               'anticipation', 'joy', 'surprise', 'trust']
```

Data Modeling

In [2]:
```
#  Python Standard Packages
import itertools
import joblib
import re
import string
import time

#  Conda Standard Packages
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.compose import ColumnTransformer, make_column_selector as selector
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, precision_score, recall_sc
                            precision_recall_curve, PrecisionRecallDisplay, make_scorer, RocCurveDisplay
from sklearn.model_selection import cross_val_score, cross_validate, \
                                    GridSearchCV, train_test_split, RandomizedSearchCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer, MinMaxScaler, \
                                  Normalizer, StandardScaler, RobustScaler
from sklearn.svm import LinearSVC

#  Third-Party Packages
import dill as pickle
import eli5
from tqdm import tqdm
```

In [33]:
```
#  Set Up Options
%precision %.3f
pd.set_option('display.float_format', '{:.2f}'.format)

'''Set Up Time Tracking Functions for Pandas'''
tqdm.pandas()
```

In [4]:
```
%%html
<style>
table {float:left}
</style>
```

I. Pre-Model Set-Up

Load Data From Pickle:

In [6]:
```
df = pd.read_pickle('./data/fulldataclean.tar.gz', compression='infer')
```

In [7]:
```
df
```

Out[7]:

	target	text	neg	neu	pos	compound	anger	disgust	fear	sadness	anticipation	joy	surprise	trust
0	1	ex wife threaten suicide recently leave wife g...	0.192	0.733	0.075	-0.949	8.000	4.000	8.000	6.000	7.000	5.000	5.000	5.000
1	0	weird get affect compliment come someone know ...	0.038	0.756	0.206	0.719	0.000	1.000	0.000	0.000	2.000	1.000	1.000	2.000
2	0	finally almost never hear bad year ever swear ...	0.259	0.665	0.076	-0.700	2.000	2.000	2.000	1.000	2.000	2.000	1.000	3.000
3	1	need help help cry hard	0.286	0.403	0.311	0.111	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000
4	1	lose hello name adam struggle year afraid past...	0.193	0.731	0.076	-0.995	18.000	10.000	27.000	26.000	20.000	6.000	5.000	9.000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
232069	0	like rock go get anything go	0.080	0.920	0.000	-0.142	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
232070	0	tell many friend lonely everything deprive pre...	0.085	0.765	0.150	0.337	1.000	1.000	1.000	1.000	0.000	0.000	0.000	1.000
232071	0	pee probably taste like salty tea someone drin...	0.000	0.839	0.161	0.361	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
232072	1	usual stuff find post sympathy pity know far b...	0.176	0.718	0.105	-0.989	9.000	6.000	7.000	10.000	3.000	1.000	1.000	3.000
232073	0	still beat first bos hollow knight fight time ...	0.263	0.685	0.052	-0.861	2.000	1.000	3.000	3.000	0.000	0.000	0.000	0.000

230072 rows × 14 columns

In [8]:

```
# Make sure no missing data

df.isna().sum()
```

Out[8]:

```
target          0
text            0
neg             0
neu             0
pos             0
compound        0
anger           0
disgust         0
fear            0
sadness         0
anticipation    0
joy             0
surprise        0
trust           0
dtype: int64
```

In [9]:

```
# All data is correct datatype:

df.dtypes
```

Out[9]:

```
target            int64
text             object
neg             float64
neu             float64
pos             float64
compound        float64
anger           float64
disgust         float64
fear            float64
sadness         float64
anticipation    float64
joy             float64
surprise        float64
trust           float64
dtype: object
```

In [11]:

```
def nonstring_data(series):

    problem_list = []
    for ind in series.index:
        if not isinstance(series.loc[ind], str):
            problem_list.append(ind)
    return problem_list
```

```
problems1 = nonstring_data(df['text'])
print(problems1)
```

```
[]
```

Define X & Y

In [13]:
```
X = df.drop(['target'], axis=1)
y = df['target']
```

In [14]:
```
X.head()
```

Out[14]:

	text	neg	neu	pos	compound	anger	disgust	fear	sadness	anticipation	joy	surprise	trust
0	ex wife threaten suicide recently leave wife g...	0.192	0.733	0.075	-0.949	8.000	4.000	8.000	6.000	7.000	5.000	5.000	5.000
1	weird get affect compliment come someone know ...	0.038	0.756	0.206	0.719	0.000	1.000	0.000	0.000	2.000	1.000	1.000	2.000
2	finally almost never hear bad year ever swear ...	0.259	0.665	0.076	-0.700	2.000	2.000	2.000	1.000	2.000	2.000	1.000	3.000
3	need help help cry hard	0.286	0.403	0.311	0.111	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000
4	lose hello name adam struggle year afraid past...	0.193	0.731	0.076	-0.995	18.000	10.000	27.000	26.000	20.000	6.000	5.000	9.000

In [15]:
```
y.head()
```

Out[15]:
```
0    1
1    0
2    0
3    1
4    1
Name: target, dtype: int64
```

Train-Test Split

- Split Ratio: 70% train, 15% val, 15% test

In [16]:
```
len(X)
```

Out[16]:
```
230072
```

In [17]:
```
print(f'{len(df)*.7:.0f}:{len(df)*.15:.0f}:{len(df)*.15:.0f}')
print('\t')
print(f'Sum: {161050+34511+34511}')
```

```
161050:34511:34511

Sum: 230072
```

In [18]:
```
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=34511, random_state=42)

X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=34511, random_state=42)
```

In [19]:
```
print(f"X - Size - Train: {len(X_train)}, Val: {len(X_val)}, Test:{len(X_test)}")

print(f"X - Perc - Train: {len(X_train)/len(X)*100:.1f}%, Val: {len(X_val)/len(df)*100:.1f}%, Test:{len(X_test),
```

```
X - Size - Train: 161050, Val: 34511, Test:34511
X - Perc - Train: 70.0%, Val: 15.0%, Test:15.0%
```

In [20]:
```
print(f"y - Size - Train: {len(y_train)}, Val: {len(y_val)}, Test:{len(y_test)}")
print(f"y - Perc - Train: {len(y_train)/len(y)*100:.1f}%, Val: {len(y_val)/len(df)*100:.1f}%, Test:{len(y_test),
```

```
y - Size - Train: 161050, Val: 34511, Test:34511
y - Perc - Train: 70.0%, Val: 15.0%, Test:15.0%
```

Functions for Metrics

1. Confusion Matrix Plotting - original source

In [21]:
```python
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion Matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [22]:
```python
class Metrics:
    def __init__(self):
        self.df = pd.DataFrame(columns=['name', 'clf', 'cv_mean', 'train', 'acc', 'prec', 'rec', 'time'])

    def get_metrics(self, name, clf, pipe, Xtr, ytr, Xval, yval):
        """Method to print metrics and return df of metrics"""

        start = time.time()
        metric_dict = {'name': name, 'clf': clf}
        pipe.fit(Xtr, ytr)
        base_cv = pd.DataFrame(cross_validate(pipe, Xtr, ytr, cv=3, return_train_score=False)).mean()
        metric_dict['cv_mean'] = base_cv[2].mean()*100

        if clf == 'rfc':
            metric_dict['train'] = pipe[clf].oob_score_*100
        else:
            yhat = pipe.predict(Xtr)
            metric_dict['train'] = accuracy_score(ytr, yhat)*100

        ypred = pipe.predict(Xval)
        metric_dict['acc'] = accuracy_score(yval, ypred)*100

        print(f"TRAIN accuracy: {metric_dict['train']: .2f} %")
        print("VAL:")
        print(f"Accuracy: {metric_dict['acc']: .2f} %")

        if clf != 'dummy':
            metric_dict['rec'] = recall_score(yval, ypred)*100
            metric_dict['prec'] = precision_score(yval, ypred)*100
            print(f"Recall: {metric_dict['rec']*100: .2f} %")
            print(f"Precision: {metric_dict['prec']: .2f} %")
            conf = confusion_matrix(yval, ypred)
            plot_confusion_matrix(conf, classes=["non-risk", "suicide risk"], normalize=True)
            print(classification_report(yval, ypred, labels=[0,1]))

        end = time.time()
        metric_dict['time'] = (f'{end-start:.1f} s')

        self.df = pd.concat([self.df, pd.DataFrame(metric_dict, index=[0])], ignore_index=True)
        return self
```

In [23]:
```python
met = Metrics()
```

In [24]:
```python
met.df
```

Out[24]:

name	clf	cv_mean	train	acc	prec	rec	time

Create Column Transformer Prep Pipeline

- Separate Pipeline needed for text and numeric (emotion/sentiment) columns
- Using MinMaxScaler to avoid negative & zero values

    - Might change to StandardScaler depending on model & normalization

In [25]:
```
# Num_Pipe includes MinMaxScaler since MNB cannot handle negative values

num_pipe = Pipeline(steps=[
    ('min', MinMaxScaler())
])

text_pipe = Pipeline(steps=[
    ("squeeze", FunctionTransformer(lambda x: x.squeeze())),
    ("tfidf", TfidfVectorizer(min_df=0.05, max_df=0.95)),
    ("toarray", FunctionTransformer(lambda x: x.toarray()))
])

CT = ColumnTransformer(transformers=[
    ('num', num_pipe, selector(dtype_include=np.number)),
    ('text', text_pipe, ['text'])
])
```

II. Data Modeling

Metrics to Focus On

1. **High Accuracy Rate**: We want a model that can successfully distinguish suicide risk cases from non-suicide risk cases
2. **High Recall Rate**: We want to *avoid* failing to detect suicide risk cases, meaning we want a high recall rate.

1. Baseline and First Simple Model

In [26]:
```
X_bas_train = X_train['text']
X_bas_val = X_val['text']

print("Data for Basic (tf-idf only) models: X_bas_")
```

```
Data for Basic (tf-idf only) models: X_bas_
```

1a. Baseline Model

In [27]:
```
base_pipe = Pipeline(steps=[
    ('tfidf', TfidfVectorizer(min_df=0.05, max_df=0.95)),
    ('dummy', DummyClassifier())
])
```

In [28]:
```
met.get_metrics('baseline', 'dummy', base_pipe, X_bas_train, y_train, X_bas_val, y_val)
```

```
TRAIN accuracy:  50.29 %
VAL:
Accuracy:  50.40 %
```

Out[28]:
```
<__main__.Metrics at 0x15f0434d0>
```

Summary of Baseline Model

- 50% accuracy in classifying suicide risk vs. non-risk
    - Expected value given equal class balance

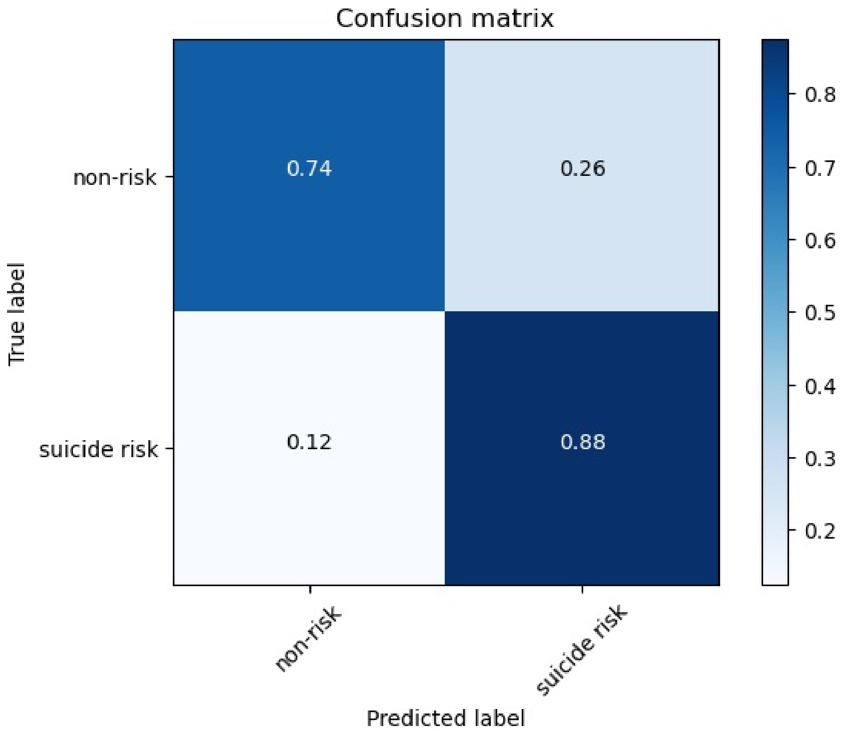
1b. First Simple Model

In [30]:
```
fsm_pipe = Pipeline(steps=[
    ('tfidf', TfidfVectorizer(min_df=0.05, max_df=0.95)),
    ('mnb', MultinomialNB())
])
```

In [31]:
```
met.get_metrics('fsm', 'mnb', fsm_pipe, X_bas_train, y_train, X_bas_val, y_val)
```

```
TRAIN accuracy:  80.47 %
VAL:
Accuracy:  80.84 %
Recall:  8750.14 %
Precision:  77.42 %
Normalized confusion matrix
[[0.74072559 0.25927441]
 [0.12498563 0.87501437]]
              precision    recall  f1-score   support

           0       0.85      0.74      0.79     17117
           1       0.77      0.88      0.82     17394

    accuracy                           0.81     34511
   macro avg       0.81      0.81      0.81     34511
weighted avg       0.81      0.81      0.81     34511
```

Out[31]: `<__main__.Metrics at 0x15f0434d0>`



In [32]:
```
met.df
```

Out[32]:

	name	clf	cv_mean	train	acc	prec	rec	time
0	baseline	dummy	50.289	50.289	50.401	NaN	NaN	16.3 s
1	fsm	mnb	80.372	80.471	80.841	77.424	87.501	16.4 s

Summary of First Simple Model

- 80.8% Accuracy in classifying suicide risk
  - Model is well-fit -- (0.4% difference between Train & Test Accuracy)
  - Quick runtime
- Recall (87.5%) > Precision (77.4%)
  - In line with goal to avoid false negatives

1c. Extracting TFIDF and stopword data

In [34]:
```
stopwords = fsm_pipe['tfidf'].stop_words_
```

In [35]:
```
stopwords = list(stopwords)
```

In [37]:
```
print(f'List of custom Stopwords: {len(stopwords)}')
print('\t')
print(stopwords[:20])
print('\t')
print(stopwords[-20:])
```

```
List of custom Stopwords: 80116

['horrify', 'hmmwv', 'homily', 'boix', 'hyposexuality', 'schism', 'steamunlocked', 'fork', 'bunnygirl', 'appear
tnly', 'diffuculty', 'sample', 'genially', 'sdead', 'hhah', 'motherfuggin', 'intuit', 'faurschou', 'flareup', '
roble']

['dadadada', 'ninny', 'coincide', 'gerninja', 'borris', 'letitia', 'borked', 'necessary', 'anxiey', 'lifestyle'
, 'meaningfull', 'optifine', 'sfriendship', 'intuitively', 'ablissful', 'hadcommanded', 'semeber', 'ourside', '
incelism', 'friggin']
```

- Custom Stopwords seem to be mostly typos, some slang or rarely used nouns

In [38]:
```
feature_words = fsm_pipe['tfidf'].get_feature_names_out()
```

In [39]:
```
df_tf_train = pd.DataFrame(
    data=(fsm_pipe['tfidf'].transform(X_bas_train)).toarray(), index=y_train.index, columns=feature_words)

df_tf_val = pd.DataFrame(
    data=(fsm_pipe['tfidf'].transform(X_bas_val)).toarray(), index=y_val.index, columns=feature_words)
```

---

2. Second Simple Model: TF-IDF and Sentiment Analysis

In [40]:
```
X_sent_train = X_train.iloc[:, 0:5]
X_sent_val = X_val.iloc[:, 0:5]

print("Data for tf-idf + sentiment models: X_Sent_")
```

```
Data for tf-idf + sentiment models: X_Sent_
```

- TF-IDF Data does not need any more preprocessing
- However, sentiment data needs to be standardized (possibly normalized in future models)

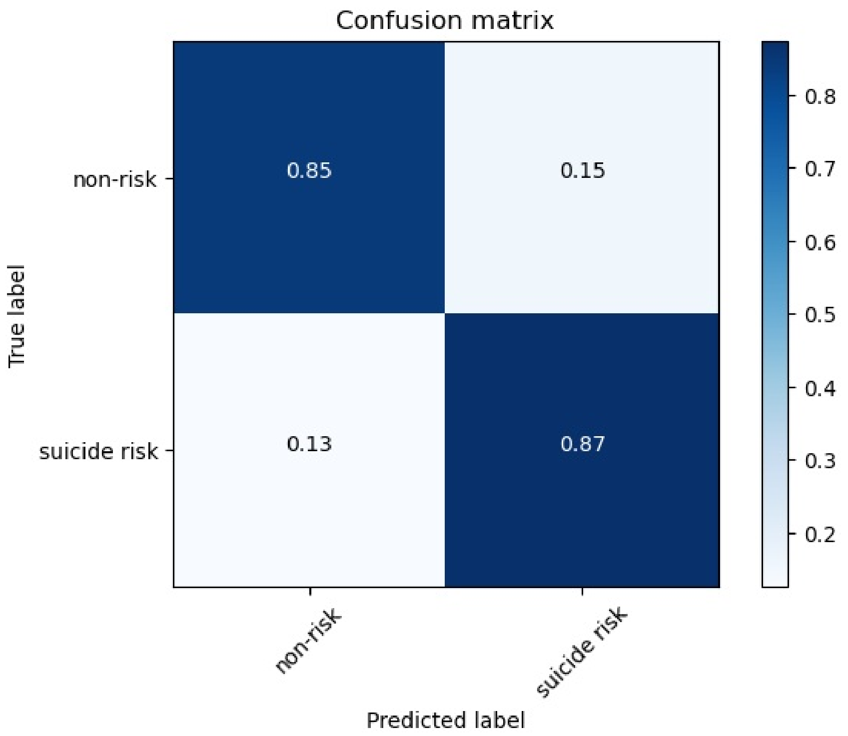
2a. Second Simple Model - MNB

In [47]:
```
ssm_pipe = Pipeline(steps=[
    ('prep', CT),
    ('mnb', MultinomialNB())
])
```

In [48]:
```
met.get_metrics('ssm', 'mnb', ssm_pipe, X_sent_train, y_train, X_sent_val, y_val)
```

```
TRAIN accuracy:  85.72 %
VAL:
Accuracy:  85.98 %
Recall:  8739.22 %
Precision:  85.17 %
Normalized confusion matrix
[[0.84535842 0.15464158]
 [0.12607796 0.87392204]]
              precision    recall  f1-score   support

           0       0.87      0.85      0.86     17117
           1       0.85      0.87      0.86     17394

    accuracy                           0.86     34511
   macro avg       0.86      0.86      0.86     34511
weighted avg       0.86      0.86      0.86     34511
```

Out[48]:
```
<__main__.Metrics at 0x15f0434d0>
```



In [38]:
```
met.df
```

Out[38]:

	name	clf	cv_mean	train	acc	prec	rec	time
0	baseline	dummy	50.289	50.289	50.401	50.401	100.000	16.4 s
1	1st_sm	mnb	80.372	80.471	80.841	77.424	87.501	16.6 s
2	2nd_sm	mnb	85.711	85.717	85.975	85.169	87.392	17.2 s

Summary of MNB - 2

- Increased Accuracy (85.98%)
  - Model is well fit (0.2% under)
  - Quick
- Recall (87.39%) higher than Precision (85.17%)

---

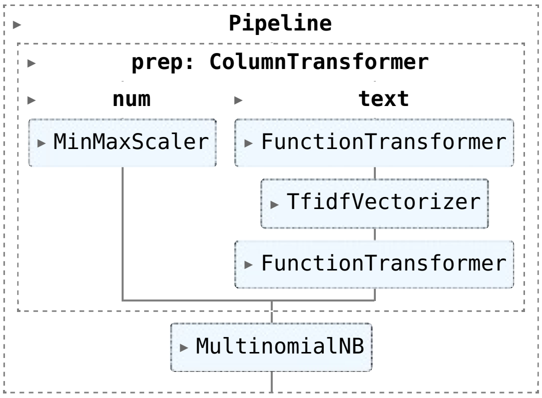
3. Full Feature Models - Comparing Classifiers

3a. MNB - Full

In [49]:
```
mnb_pipe = Pipeline(steps=[
    ('prep', CT),
    ('mnb', MultinomialNB())
])
```

In [50]:
```
mnb_pipe
```

Out[50]:

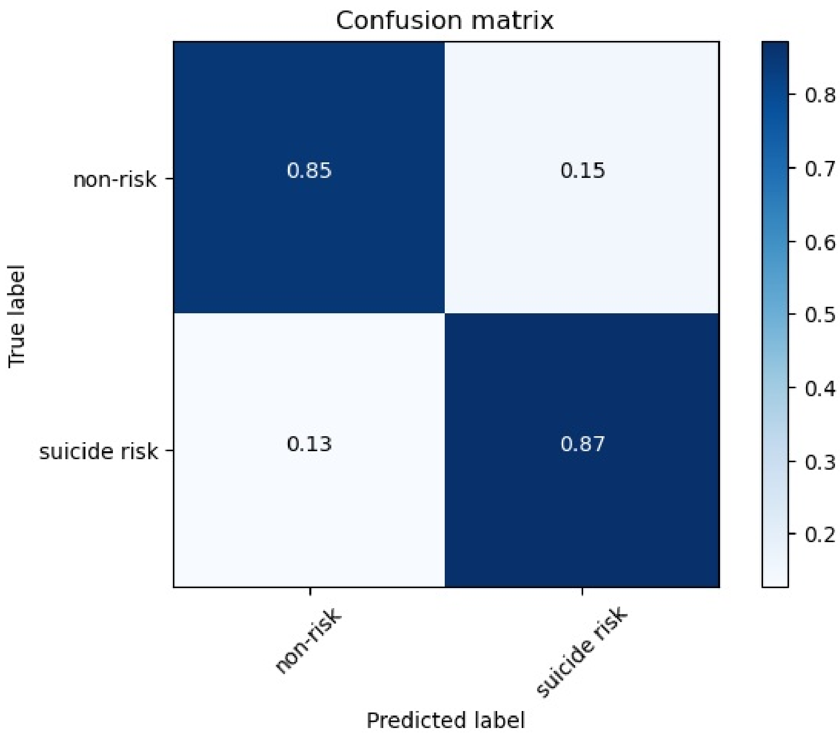


In [51]:

```
met.get_metrics('full', 'mnb', mnb_pipe, X_train, y_train, X_val, y_val)
```

```
TRAIN accuracy:  85.77 %
VAL:
Accuracy:  86.08 %
Recall:  8719.10 %
Precision:  85.48 %
Normalized confusion matrix
[[0.84944792 0.15055208]
 [0.12809015 0.87190985]]
              precision    recall  f1-score   support

           0       0.87      0.85      0.86     17117
           1       0.85      0.87      0.86     17394

    accuracy                           0.86     34511
   macro avg       0.86      0.86      0.86     34511
weighted avg       0.86      0.86      0.86     34511
```

Out[51]: <__main__.Metrics at 0x15f0434d0>



In [52]:

```
met.df
```

Out[52]:

	name	clf	cv_mean	train	acc	prec	rec	time
0	baseline	dummy	50.29	50.29	50.40	NaN	NaN	16.3 s
1	fsm	mnb	80.37	80.47	80.84	77.42	87.50	16.4 s
2	ssm	mnb	85.71	85.72	85.98	85.17	87.39	16.8 s
3	full	mnb	85.76	85.77	86.08	85.48	87.19	17.2 s

- Adding Emotion did very little to the MNB model
  - Slight accuracy improvement (0.1%)
  - Recall slightly worse (0.2%)
  - Slightly underfit (0.3%)

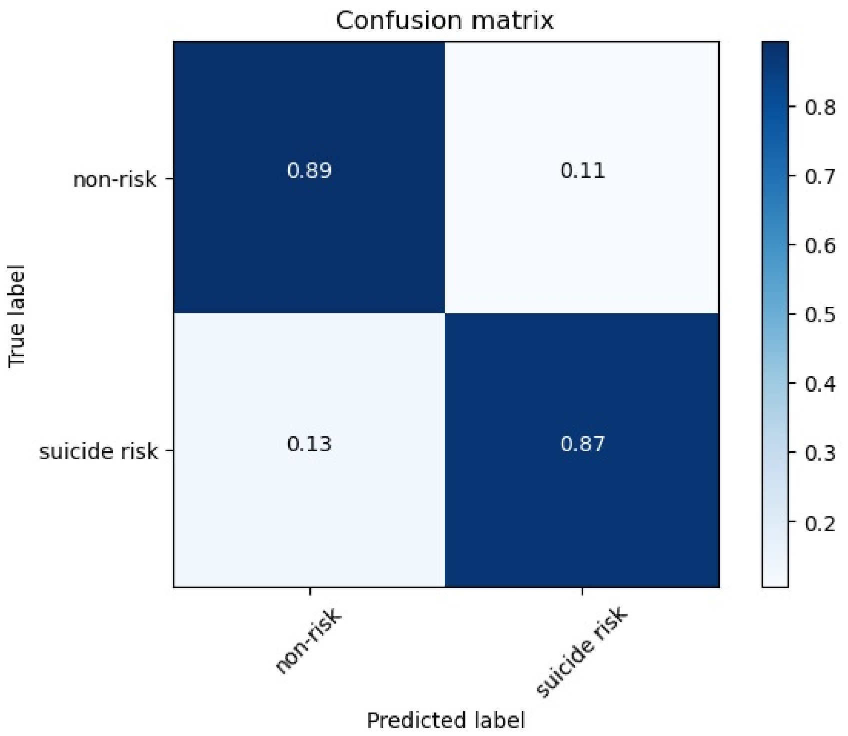
3b. Full - RFC

In [53]:
```
rfc_pipe = Pipeline(steps=[
    ('prep', CT),
    ('rfc', RandomForestClassifier(oob_score=True, n_jobs= -1, random_state=42))
])
```

In [54]:
```
met.get_metrics('full', 'rfc', rfc_pipe, X_train, y_train, X_val, y_val)
```

```
TRAIN accuracy:  88.26 %
VAL:
Accuracy:  88.37 %
Recall:  8738.65 %
Precision:  89.32 %
Normalized confusion matrix
[[0.8937898  0.1062102 ]
 [0.12613545 0.87386455]]
              precision    recall  f1-score   support

           0       0.87      0.89      0.88     17117
           1       0.89      0.87      0.88     17394

    accuracy                           0.88     34511
   macro avg       0.88      0.88      0.88     34511
weighted avg       0.88      0.88      0.88     34511
```

Out[54]: `<__main__.Metrics at 0x15f0434d0>`



In [55]:
```
met.df
```

Out[55]:

	name	clf	cv_mean	train	acc	prec	rec	time
0	baseline	dummy	50.29	50.29	50.40	NaN	NaN	16.3 s
1	fsm	mnb	80.37	80.47	80.84	77.42	87.50	16.4 s
2	ssm	mnb	85.71	85.72	85.98	85.17	87.39	16.8 s
3	full	mnb	85.76	85.77	86.08	85.48	87.19	17.2 s
4	full	rfc	88.26	88.26	88.37	89.32	87.39	43.8 s

- Higher accuracy than MNB (88.3% vs. 86.1%)
- Well fit (0.1% under)
- However:
    - Much longer to run
    - Fails to minimize false negatives (precision > recall)
- STILL, recall is slightly better than MNB (87.4% vs. 87.2%)

WINNER: RFC

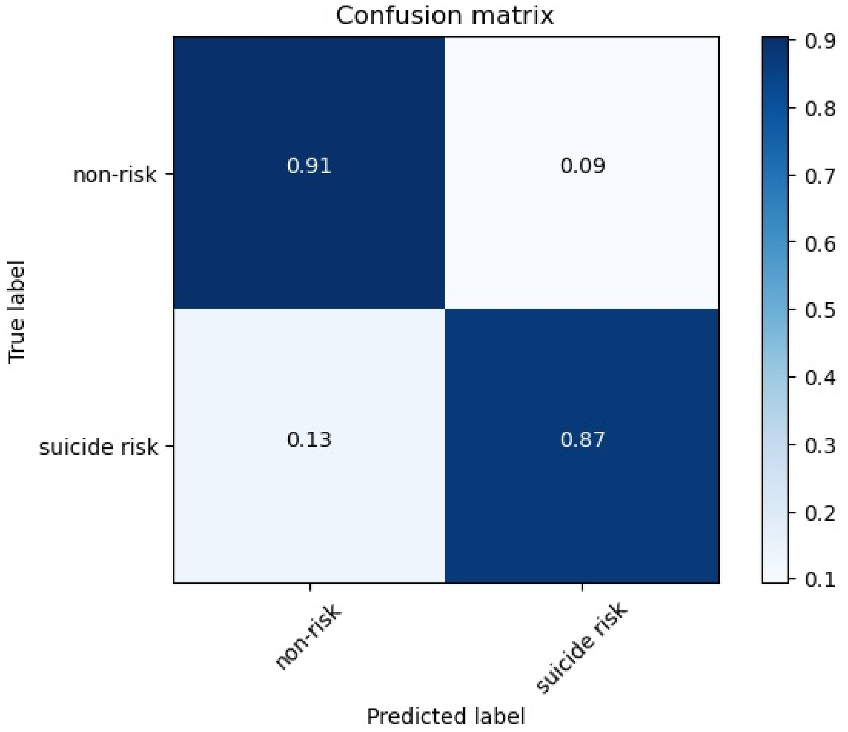
3c. Logistic Regression

In [56]:
```
lr_pipe = Pipeline(steps=[
    ('prep', CT),
    ('lr', LogisticRegression(
        solver='saga', random_state=42))
])
```

In [57]:
```
met.get_metrics('full', 'lr', lr_pipe, X_train, y_train, X_val, y_val)
```

```
TRAIN accuracy:  88.93 %
VAL:
Accuracy:  88.87 %
Recall:  8721.97 %
Precision:  90.36 %
Normalized confusion matrix
[[0.90541567 0.09458433]
 [0.12780269 0.87219731]]
              precision    recall  f1-score   support

           0       0.87      0.91      0.89     17117
           1       0.90      0.87      0.89     17394

    accuracy                           0.89     34511
   macro avg       0.89      0.89      0.89     34511
weighted avg       0.89      0.89      0.89     34511
```

Out[57]:
```
<__main__.Metrics at 0x15f0434d0>
```



In [58]:
```
met.df
```

Out[58]:

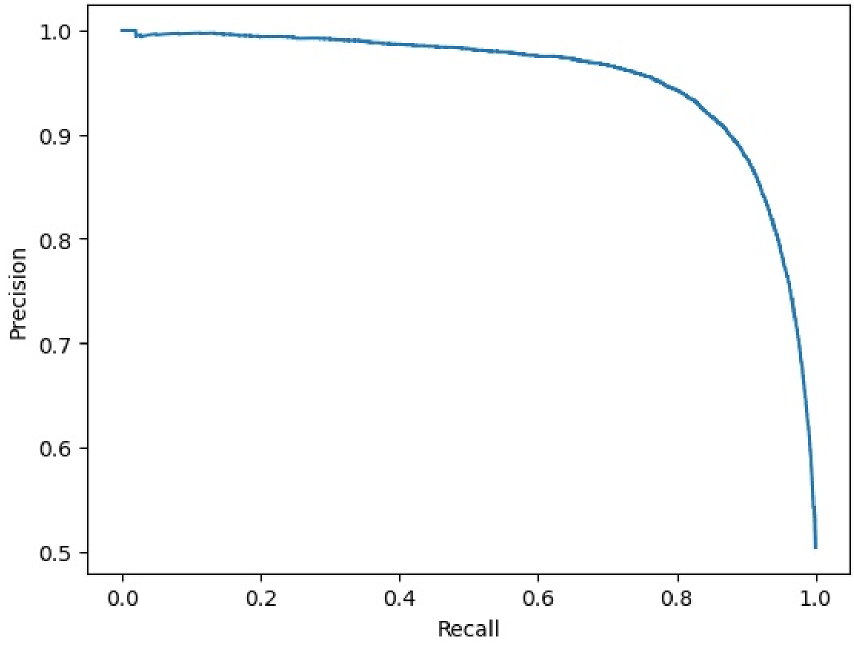
	name	clf	cv_mean	train	acc	prec	rec	time
0	baseline	dummy	50.29	50.29	50.40	NaN	NaN	16.3 s
1	fsm	mnb	80.37	80.47	80.84	77.42	87.50	16.4 s
2	ssm	mnb	85.71	85.72	85.98	85.17	87.39	16.8 s
3	full	mnb	85.76	85.77	86.08	85.48	87.19	17.2 s
4	full	rfc	88.26	88.26	88.37	89.32	87.39	43.8 s
5	full	lr	88.86	88.93	88.87	90.36	87.22	31.8 s

- Higher accuracy than RFC or MNB (0.50%)
- Lower recall score than RFC (0.17%) & greater margin between precision and recall (3.14 vs. 1.93)

- However, margin can easily be altered by changing threshold:

In [59]:
```
y_pred_proba = lr_pipe.predict_proba(X_val)
lr_prc = pd.DataFrame(precision_recall_curve(y_val, y_pred_proba[:, 1])).T
```

In [61]:
```
lr_prc.columns = ['precision', 'recall', 'threshold']

PrecisionRecallDisplay(precision=lr_prc['precision'], recall=lr_prc['recall']).plot();
```



In [63]:
```
lr_prc.mask(lr_prc['recall']<0.9).dropna().sort_values(by='precision', ascending=False)
```

Out[63]:

	precision	recall	threshold
16251	0.88	0.90	0.40
16250	0.88	0.90	0.40
16248	0.88	0.90	0.40
16249	0.88	0.90	0.40
16246	0.88	0.90	0.40
...	...	...	...
4	0.50	1.00	0.00
3	0.50	1.00	0.00
2	0.50	1.00	0.00
1	0.50	1.00	0.00
0	0.50	1.00	0.00

16252 rows × 3 columns

- Lowering threshold will increase recall, so goal of hyperparameter tuning should be to increase AUC

WINNER: Logistic Regression Model

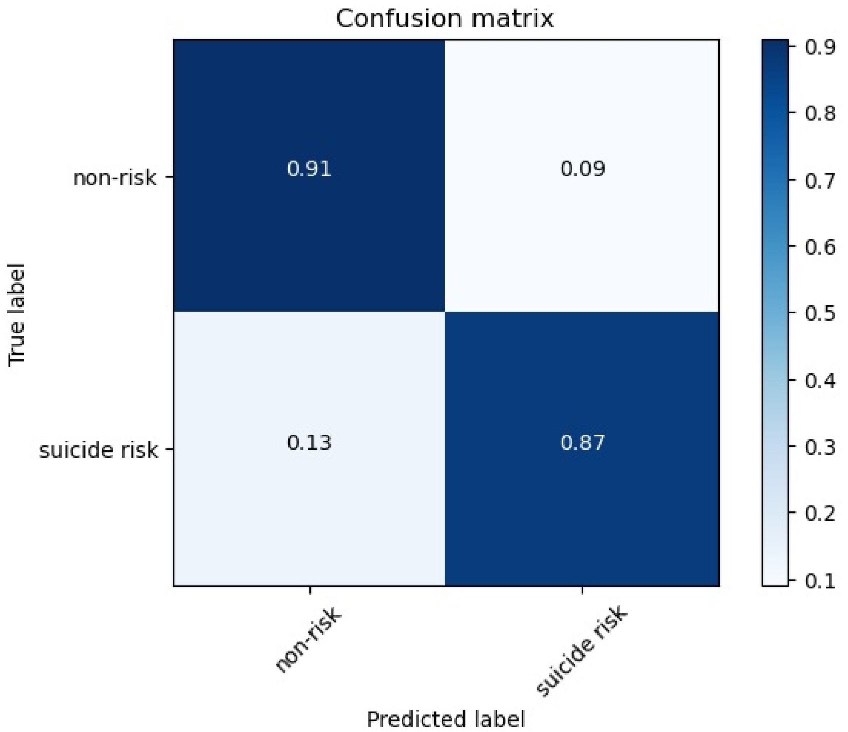
3d. SVM

In [64]:
```
svm_pipe = Pipeline(steps=[
    ('prep', CT),
    ('svm', LinearSVC(dual=False))
])
```

In [65]:
```
met.get_metrics('full', 'svm', svm_pipe, X_train, y_train, X_val, y_val)
```

```
TRAIN accuracy:  88.98 %
VAL:
Accuracy:  88.90 %
Recall:  8684.60 %
Precision:  90.73 %
Normalized confusion matrix
[[0.90979728 0.09020272]
 [0.13153961 0.86846039]]
              precision    recall  f1-score   support

           0       0.87      0.91      0.89     17117
           1       0.91      0.87      0.89     17394

    accuracy                           0.89     34511
   macro avg       0.89      0.89      0.89     34511
weighted avg       0.89      0.89      0.89     34511
```

Out[65]: <__main__.Metrics at 0x15f0434d0>



In [66]:
```
met.df
```

Out[66]:

	name	clf	cv_mean	train	acc	prec	rec	time
0	baseline	dummy	50.29	50.29	50.40	NaN	NaN	16.3 s
1	fsm	mnb	80.37	80.47	80.84	77.42	87.50	16.4 s
2	ssm	mnb	85.71	85.72	85.98	85.17	87.39	16.8 s
3	full	mnb	85.76	85.77	86.08	85.48	87.19	17.2 s
4	full	rfc	88.26	88.26	88.37	89.32	87.39	43.8 s
5	full	lr	88.86	88.93	88.87	90.36	87.22	31.8 s
6	full	svm	88.92	88.98	88.90	90.73	86.85	21.5 s

- While SVM has the highest accuracy, it also has the lowest recall rate.
- Logistic Regression is probably a better option, since accuracy is not that different between LR and SVM

Best Full Feature Model:

LR (88.9% Accuracy, 87.2% Recall, -3.14% RvP)

- Can change threshold to increase recall over precision

In [ ]:
```
# joblib.dump(lr_pipe, './models/3c_lr_pipe.joblib')
```

Out[ ]: ['./models/3c_lr_pipe.joblib']

III. Hyperparameter Tuning

Function for Grid Search Metrics

In [71]:
```
def get_grid_val_metrics(metrics, name, clf, pipe, Xtr, ytr, Xval, yval):

    start = time.time()

    met_dict = {'name':name, 'clf':clf}

    pipe.fit(X_train, y_train)
    yhat = pipe.predict(Xtr)
    met_dict['train'] = accuracy_score(ytr, yhat)*100

    ypred = pipe.predict(Xval)
    met_dict['acc'] = accuracy_score(yval, ypred)*100
    met_dict['prec'] = precision_score(yval, ypred)*100
    met_dict['rec'] = recall_score(yval, ypred)*100

    ypr = pipe.predict_proba(Xval)
    met_dict['roc_auc'] = roc_auc_score(yval, ypr[:,1])*100

    metrics = pd.concat([metrics, pd.DataFrame(met_dict, index=[0])], ignore_index=True)

    return metrics
```

In [72]:
```
metrics = pd.DataFrame()
```

In [73]:
```
metrics = get_grid_val_metrics(metrics, 'lr_pipe', 'lr', lr_pipe, X_train, y_train, X_val, y_val)
```

In [74]:
```
metrics
```

Out[74]:

	name	clf	train	acc	prec	rec	roc_auc
0	lr_pipe	lr	88.93	88.87	90.36	87.22	95.14

---

1. Grid Search on Prep CT (Num Scaler and TF-IDF)

1. Type of Scaler & Normalization used on Numbers
2. TF-IDF min_df and max_df

In [75]:
```
lr_scoring = {'accuracy': make_scorer(accuracy_score),
              'roc_auc': make_scorer(roc_auc_score, needs_proba=True)}
```

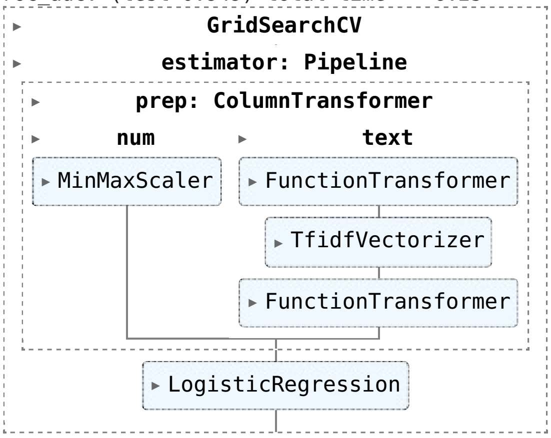
In [76]:
```
numprep_params = {'prep__num__steps': [
        [('min', MinMaxScaler())],
        [('ss', StandardScaler()),
         ('norm', Normalizer(norm='l2'))],
        [('ss', StandardScaler()),
         ('norm', Normalizer(norm='l1'))],
        [('rs', RobustScaler()),
         ('norm', Normalizer(norm='l2'))],
        [('rs', RobustScaler()),
         ('norm', Normalizer(norm='l1'))]
    ]
}
```

In [77]:
```
num_prep = GridSearchCV(
    estimator=lr_pipe,
    param_grid = numprep_params,
    scoring=lr_scoring, refit='roc_auc',
    cv=3, verbose=3)
```

In [78]:
```
num_prep.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 5 candidates, totalling 15 fits
[CV 1/3] END prep__num__steps=[('min', MinMaxScaler())]; accuracy: (test=0.890) roc_auc: (test=0.952) total tim
e=   7.8s
[CV 2/3] END prep__num__steps=[('min', MinMaxScaler())]; accuracy: (test=0.889) roc_auc: (test=0.952) total tim
e=   7.5s
[CV 3/3] END prep__num__steps=[('min', MinMaxScaler())]; accuracy: (test=0.887) roc_auc: (test=0.949) total tim
e=   7.5s
[CV 1/3] END prep__num__steps=[('ss', StandardScaler()), ('norm', Normalizer())]; accuracy: (test=0.889) roc_au
c: (test=0.952) total time=   6.4s
[CV 2/3] END prep__num__steps=[('ss', StandardScaler()), ('norm', Normalizer())]; accuracy: (test=0.890) roc_au
c: (test=0.952) total time=   6.2s
[CV 3/3] END prep__num__steps=[('ss', StandardScaler()), ('norm', Normalizer())]; accuracy: (test=0.888) roc_au
c: (test=0.950) total time=   6.3s
[CV 1/3] END prep__num__steps=[('ss', StandardScaler()), ('norm', Normalizer(norm='l1'))]; accuracy: (test=0.88
9) roc_auc: (test=0.952) total time=   6.5s
[CV 2/3] END prep__num__steps=[('ss', StandardScaler()), ('norm', Normalizer(norm='l1'))]; accuracy: (test=0.89
0) roc_auc: (test=0.952) total time=   6.3s
[CV 3/3] END prep__num__steps=[('ss', StandardScaler()), ('norm', Normalizer(norm='l1'))]; accuracy: (test=0.88
8) roc_auc: (test=0.950) total time=   6.1s
[CV 1/3] END prep__num__steps=[('rs', RobustScaler()), ('norm', Normalizer())]; accuracy: (test=0.889) roc_auc:
(test=0.952) total time=   6.3s
[CV 2/3] END prep__num__steps=[('rs', RobustScaler()), ('norm', Normalizer())]; accuracy: (test=0.890) roc_auc:
(test=0.952) total time=   6.3s
[CV 3/3] END prep__num__steps=[('rs', RobustScaler()), ('norm', Normalizer())]; accuracy: (test=0.887) roc_auc:
(test=0.950) total time=   6.3s
[CV 1/3] END prep__num__steps=[('rs', RobustScaler()), ('norm', Normalizer(norm='l1'))]; accuracy: (test=0.889)
roc_auc: (test=0.952) total time=   6.5s
[CV 2/3] END prep__num__steps=[('rs', RobustScaler()), ('norm', Normalizer(norm='l1'))]; accuracy: (test=0.889)
roc_auc: (test=0.952) total time=   6.3s
[CV 3/3] END prep__num__steps=[('rs', RobustScaler()), ('norm', Normalizer(norm='l1'))]; accuracy: (test=0.887)
roc_auc: (test=0.949) total time=   6.2s
```

Out[78]:



In [88]:

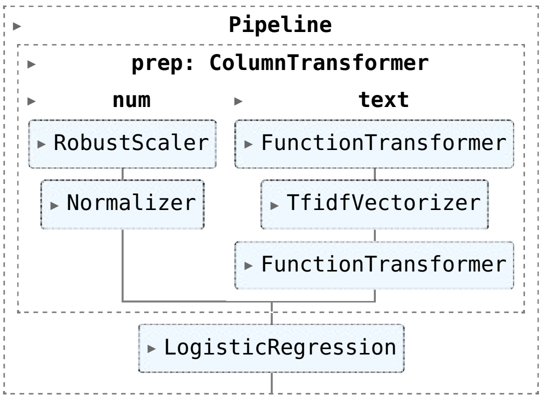
```
print(num_prep.best_params_, num_prep.best_score_)
```

```
{'prep__num__steps': [('rs', RobustScaler()), ('norm', Normalizer())]} 0.9512438199032411
```

In [83]:

```
num_prep_pipe = num_prep.best_estimator_
num_prep_pipe
```

Out[83]:



In [84]:

```
metrics = get_grid_val_metrics(metrics, 'num_prep_pipe', 'lr', num_prep_pipe, X_train, y_train, X_val, y_val)
```

In [85]:

```
metrics
```

Out[85]:

	name	clf	train	acc	prec	rec	roc_auc
0	lr_pipe	lr	88.93	88.87	90.36	87.22	95.14
1	num_prep_pipe	lr	88.92	88.87	90.42	87.14	95.19

- Even though accuracy is unchanged and recall *decreases*, the AUC increases, which will help the recall rate when the threshold is changed.

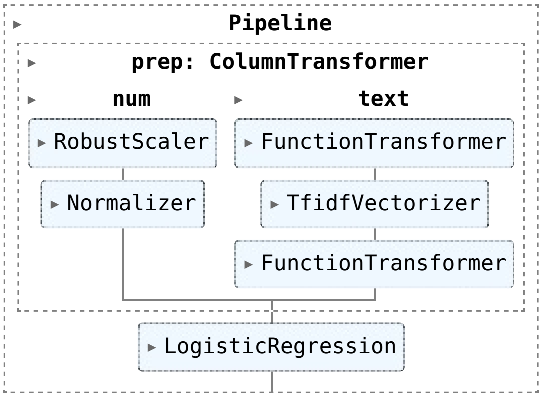
Manually Testing tf-idf min_df/max_df metric

- Throws an error in GridSearch, possibly because of custom function transfomers used

In [86]:
```
text_pipe = num_prep_pipe
```

In [87]:
```
text_pipe.set_params(prep__text__tfidf__max_df=0.99, prep__text__tfidf__min_df=0.01)
```

Out[87]:



In [89]:
```
metrics = get_grid_val_metrics(metrics, 'maxdf_99', 'lr', text_pipe, X_train, y_train, X_val, y_val)
```

In [90]:
```
metrics
```

Out[90]:

	name	clf	train	acc	prec	rec	roc_auc
0	lr_pipe	lr	88.93	88.87	90.36	87.22	95.14
1	num_prep_pipe	lr	88.92	88.87	90.42	87.14	95.19
2	maxdf_99	lr	91.76	91.39	92.68	90.03	97.04

- Reducing df cutoff increases all metrics by 2 to 3%

In [92]:
```
words_added = len(lr_pipe['prep'].named_transformers_['text']['tfidf'].stop_words_) -\
              alen(text_pipe['prep'].named_transformers_['text']['tfidf'].stop_words_)

print(f"New model includes {words_added} words that were previously removed as stopwords.")
```

```
New model includes 564 words that were previously removed as stopwords.
```

---

2. Logistic Regression - Hyperparameter Tuning

In [110…
```
# First, testing different L1 ratios with "ElasticNet" option on

text_pipe.set_params(**{'lr__penalty': 'elasticnet'})

lr_params = {
    'lr__l1_ratio': [0.0, 0.25, 0.75, 1.0],
    'lr__C': [1, 1000],
    'lr__tol': [0.0001, 0.000001]
}
```

In [111…
```
lr_gs1 = GridSearchCV(
    estimator=text_pipe,
    param_grid=lr_params,
    scoring=lr_scoring, refit='roc_auc',
    cv=3, verbose=3)
```

In [112…
```
start = time.time()

lr_gs1.fit(X_train, y_train)

end = time.time()

print('\n')
print(f'Time to fit: {(end - start) // 60} min {(end - start)%60:.0f} sec')
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits
```

```
[CV 1/3] END lr__C=1, lr__l1_ratio=0.0, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total time
=  12.1s
[CV 2/3] END lr__C=1, lr__l1_ratio=0.0, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total time
=  11.9s
[CV 3/3] END lr__C=1, lr__l1_ratio=0.0, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total time
=  11.9s
[CV 1/3] END lr__C=1, lr__l1_ratio=0.0, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total time=
15.6s
[CV 2/3] END lr__C=1, lr__l1_ratio=0.0, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total time=
15.7s
[CV 3/3] END lr__C=1, lr__l1_ratio=0.0, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total time=
16.2s
[CV 1/3] END lr__C=1, lr__l1_ratio=0.25, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total tim
e=  14.7s
[CV 2/3] END lr__C=1, lr__l1_ratio=0.25, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total tim
e=  14.7s
[CV 3/3] END lr__C=1, lr__l1_ratio=0.25, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total tim
e=  16.1s
[CV 1/3] END lr__C=1, lr__l1_ratio=0.25, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total time
=  19.4s
[CV 2/3] END lr__C=1, lr__l1_ratio=0.25, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total time
=  19.7s
[CV 3/3] END lr__C=1, lr__l1_ratio=0.25, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total time
=  20.1s
[CV 1/3] END lr__C=1, lr__l1_ratio=0.75, lr__tol=0.0001; accuracy: (test=0.917) roc_auc: (test=0.971) total tim
e=  15.9s
[CV 2/3] END lr__C=1, lr__l1_ratio=0.75, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total tim
e=  15.5s
[CV 3/3] END lr__C=1, lr__l1_ratio=0.75, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total tim
e=  16.1s
[CV 1/3] END lr__C=1, lr__l1_ratio=0.75, lr__tol=1e-06; accuracy: (test=0.917) roc_auc: (test=0.971) total time
=  20.3s
[CV 2/3] END lr__C=1, lr__l1_ratio=0.75, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total time
=  21.5s
[CV 3/3] END lr__C=1, lr__l1_ratio=0.75, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total time
=  21.8s
[CV 1/3] END lr__C=1, lr__l1_ratio=1.0, lr__tol=0.0001; accuracy: (test=0.917) roc_auc: (test=0.971) total time
=  16.2s
[CV 2/3] END lr__C=1, lr__l1_ratio=1.0, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total time
=  16.3s
[CV 3/3] END lr__C=1, lr__l1_ratio=1.0, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total time
=  16.4s
[CV 1/3] END lr__C=1, lr__l1_ratio=1.0, lr__tol=1e-06; accuracy: (test=0.917) roc_auc: (test=0.971) total time=
23.5s
[CV 2/3] END lr__C=1, lr__l1_ratio=1.0, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total time=
23.4s
[CV 3/3] END lr__C=1, lr__l1_ratio=1.0, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total time=
22.1s
[CV 1/3] END lr__C=1000, lr__l1_ratio=0.0, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  12.4s
[CV 2/3] END lr__C=1000, lr__l1_ratio=0.0, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  13.0s
[CV 3/3] END lr__C=1000, lr__l1_ratio=0.0, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total t
ime=  13.1s
[CV 1/3] END lr__C=1000, lr__l1_ratio=0.0, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total ti
me=  18.0s
[CV 2/3] END lr__C=1000, lr__l1_ratio=0.0, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total ti
me=  17.9s
[CV 3/3] END lr__C=1000, lr__l1_ratio=0.0, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total ti
me=  17.0s
[CV 1/3] END lr__C=1000, lr__l1_ratio=0.25, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total
time=  15.2s
[CV 2/3] END lr__C=1000, lr__l1_ratio=0.25, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total
time=  15.9s
[CV 3/3] END lr__C=1000, lr__l1_ratio=0.25, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total
time=  15.9s
[CV 1/3] END lr__C=1000, lr__l1_ratio=0.25, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  22.1s
[CV 2/3] END lr__C=1000, lr__l1_ratio=0.25, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  22.0s
[CV 3/3] END lr__C=1000, lr__l1_ratio=0.25, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total t
ime=  21.3s
[CV 1/3] END lr__C=1000, lr__l1_ratio=0.75, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total
time=  16.0s
[CV 2/3] END lr__C=1000, lr__l1_ratio=0.75, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total
time=  16.7s
[CV 3/3] END lr__C=1000, lr__l1_ratio=0.75, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total
time=  16.4s
[CV 1/3] END lr__C=1000, lr__l1_ratio=0.75, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  22.8s
[CV 2/3] END lr__C=1000, lr__l1_ratio=0.75, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  22.9s
[CV 3/3] END lr__C=1000, lr__l1_ratio=0.75, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total t
```

```
ime=  22.0s
[CV 1/3] END lr__C=1000, lr__l1_ratio=1.0, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  15.8s
[CV 2/3] END lr__C=1000, lr__l1_ratio=1.0, lr__tol=0.0001; accuracy: (test=0.916) roc_auc: (test=0.971) total t
ime=  16.3s
[CV 3/3] END lr__C=1000, lr__l1_ratio=1.0, lr__tol=0.0001; accuracy: (test=0.914) roc_auc: (test=0.969) total t
ime=  16.4s
[CV 1/3] END lr__C=1000, lr__l1_ratio=1.0, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total ti
me=  23.4s
[CV 2/3] END lr__C=1000, lr__l1_ratio=1.0, lr__tol=1e-06; accuracy: (test=0.916) roc_auc: (test=0.971) total ti
me=  23.3s
[CV 3/3] END lr__C=1000, lr__l1_ratio=1.0, lr__tol=1e-06; accuracy: (test=0.914) roc_auc: (test=0.969) total ti
me=  22.5s


Time to fit: 14.0 min 44 sec
```

In [117…
```
print(f'Best ROC-AUC: {lr_gs1.best_score_*100:.2f}, {lr_gs1.best_params_}')
```

```
Best ROC-AUC: 97.02, {'lr__C': 1, 'lr__l1_ratio': 0.75, 'lr__tol': 1e-06}
```

- Across all combinations: very little difference in accuracy and ROC-AUC Score
- Even though the "best_param" for tolerance is 1e-6, not much difference between 1e-4 and 1e-6 for tolerance.
  - Tolerance = 1e-4 is preferable because it's faster

- Best Parameters:
  - C = 1
  - L1_Ratio = 0.75
  - Tol = 1e-4 (speed)

In [120…
```
lr_grid_pipe = text_pipe
lr_grid_pipe.set_params(**{'lr__l1_ratio': 0.75})
lr_grid_pipe['lr'].get_params()
```

Out[120]:
```
{'C': 1.000,
 'class_weight': None,
 'dual': False,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'l1_ratio': 0.750,
 'max_iter': 100,
 'multi_class': 'auto',
 'n_jobs': None,
 'penalty': 'elasticnet',
 'random_state': 42,
 'solver': 'saga',
 'tol': 0.000,
 'verbose': 0,
 'warm_start': False}
```

In [122…
```
metrics = get_grid_val_metrics(metrics, 'l1_ratio_75', 'lr', lr_grid_pipe, X_train, y_train, X_val, y_val)
```

In [123…
```
metrics
```

Out[123]:

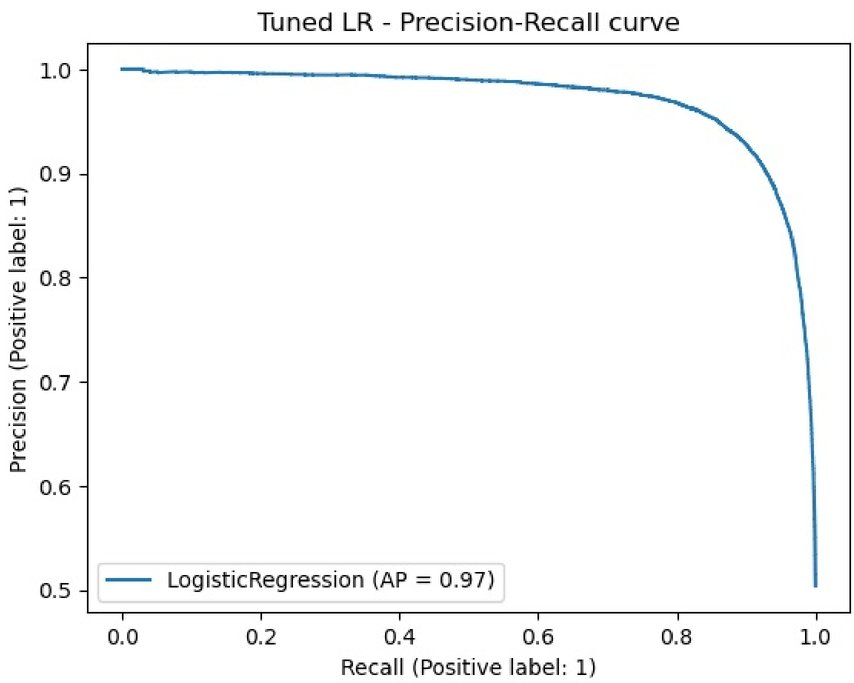
	name	clf	train	acc	prec	rec	roc_auc
0	lr_pipe	lr	88.93	88.87	90.36	87.22	95.14
1	num_prep_pipe	lr	88.92	88.87	90.42	87.14	95.19
2	maxdf_99	lr	91.76	91.39	92.68	90.03	97.04
3	l1_ratio_75	lr	91.81	91.46	92.72	90.13	97.05

- Very little improvements in accuracy and roc_auc with hyperparameter tuning-- but still an improvement!

3. Altering Threshold to Increase Recall

In [124…
```
y_pred_proba = lr_grid_pipe.predict_proba(X_val)
prc = pd.DataFrame(precision_recall_curve(y_val, y_pred_proba[:,1])).T
```

In [125…
```
display = PrecisionRecallDisplay.from_estimator(lr_grid_pipe, X_val, y_val, name="LogisticRegression")
display.ax_.set_title("Tuned LR - Precision-Recall curve");
```



In [126…
```
prc.columns = ['precision', 'recall', 'threshold']
```

In [127…
```
prc.mask(prc['recall']<0.92).dropna().sort_values(by='precision', ascending=False)[:5]
```

Out[127]:

	precision	recall	threshold
16771	0.91	0.92	0.41
16767	0.91	0.92	0.41
16768	0.91	0.92	0.41
16769	0.91	0.92	0.41
16770	0.91	0.92	0.41

- Testing a threshold of 0.4:

In [128…
```
y_pred = np.where(lr_grid_pipe.predict_proba(X_val)[:,1] > 0.400, 1, 0)
```

In [129…
```
print(accuracy_score(y_val, y_pred), precision_score(y_val, y_pred), recall_score(y_val, y_pred))
```

```
0.9137956014024514 0.9074314778185928 0.9231344141658043
```

In [135…
```
metrics
```

Out[135]:

	name	clf	train	acc	prec	rec	roc_auc
0	lr_pipe	lr	88.93	88.87	90.36	87.22	95.14
1	num_prep_pipe	lr	88.92	88.87	90.42	87.14	95.19
2	maxdf_99	lr	91.76	91.39	92.68	90.03	97.04
3	l1_ratio_75	lr	91.81	91.46	92.72	90.13	97.05

In [137…
```
final_mets = {
    'name':'threshold',
    'clf':'lr',
    'train':'NA',
    'acc': accuracy_score(y_val, y_pred)*100,
    'prec': precision_score(y_val, y_pred)*100,
    'rec': recall_score(y_val, y_pred)*100,
    'roc_auc': roc_auc_score(y_val, y_pred)*100
}
```

In [141…
```
metrics = pd.concat([metrics, pd.DataFrame(final_mets, index=[0])], ignore_index=True)
```

In [142…
```
metrics
```

Out[142]:

	name	clf	train	acc	prec	rec	roc_auc
0	lr_pipe	lr	88.93	88.87	90.36	87.22	95.14
1	num_prep_pipe	lr	88.92	88.87	90.42	87.14	95.19
2	maxdf_99	lr	91.76	91.39	92.68	90.03	97.04
3	l1_ratio_75	lr	91.81	91.46	92.72	90.13	97.05
4	threshold	lr	NA	91.38	90.74	92.31	91.37

- Using a threshold of 0.400 increases recall over precision, which lowers accuracy a little bit, but ultimately boosts recall.

Saving Final Model Pipeline

In [143…

```
final_pipe = lr_grid_pipe
```

In [144…

```
joblib.dump(final_pipe, './streamlit/final_pipe.joblib')
```

Out[144]:

```
['./streamlit/final_pipe.joblib']
```

IV. Final Model Results

Predicting X_test with Threshold = 0.4

In [145…

```
y_hat = np.where(final_pipe.predict_proba(X_train)[:,1] > 0.400, 1, 0)
```

In [146…

```
y_pred = np.where(final_pipe.predict_proba(X_test)[:,1] > 0.400, 1, 0)
```

In [159…

```
test_metrics = {}
test_metrics['train'] = accuracy_score(y_train, y_hat)*100
test_metrics['accuracy'] = accuracy_score(y_test, y_pred)*100
test_metrics['precision'] = precision_score(y_test, y_pred)*100
test_metrics['recall'] = recall_score(y_test, y_pred)*100
test_metrics['roc_auc'] = roc_auc_score(y_test, y_pred)*100
```

In [163…

```
print(f'Training Accuracy Score: {accuracy_score(y_train, y_hat)*100:.2f} %')
print('\t')
print("TEST SCORES:")
print(f"Accuracy: {accuracy_score(y_test, y_pred)*100: .2f} %")
print(f"Recall: {recall_score(y_test, y_pred)*100: .2f} %")
print(f"Precision: {precision_score(y_test, y_pred)*100: .2f} %")
print(f"ROC AUC: {roc_auc_score(y_test, y_pred)*100: .2f} %")
```

```
Training Accuracy Score: 91.67 %

TEST SCORES:
Accuracy:  91.57 %
Recall:  92.43 %
Precision:  91.05 %
ROC AUC:  91.55 %
```

In [164…

```
print(classification_report(y_test, y_pred, labels=[0,1]))
```

```
              precision    recall  f1-score   support

           0       0.92      0.91      0.91     17034
           1       0.91      0.92      0.92     17477

    accuracy                           0.92     34511
   macro avg       0.92      0.92      0.92     34511
weighted avg       0.92      0.92      0.92     34511
```

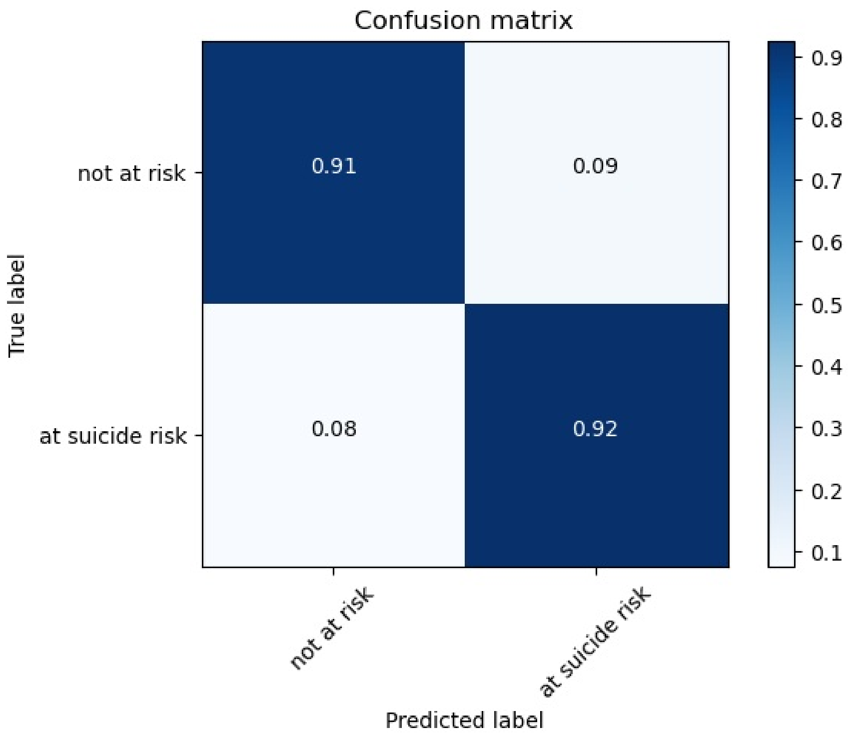
In [165…

```
conf = confusion_matrix(y_test, y_pred)
```

In [166…

```
plot_confusion_matrix(conf,
                      classes=['not at risk', 'at suicide risk'],
                      normalize=True)
```

```
Normalized confusion matrix
[[0.90677469 0.09322531]
 [0.07569949 0.92430051]]
```

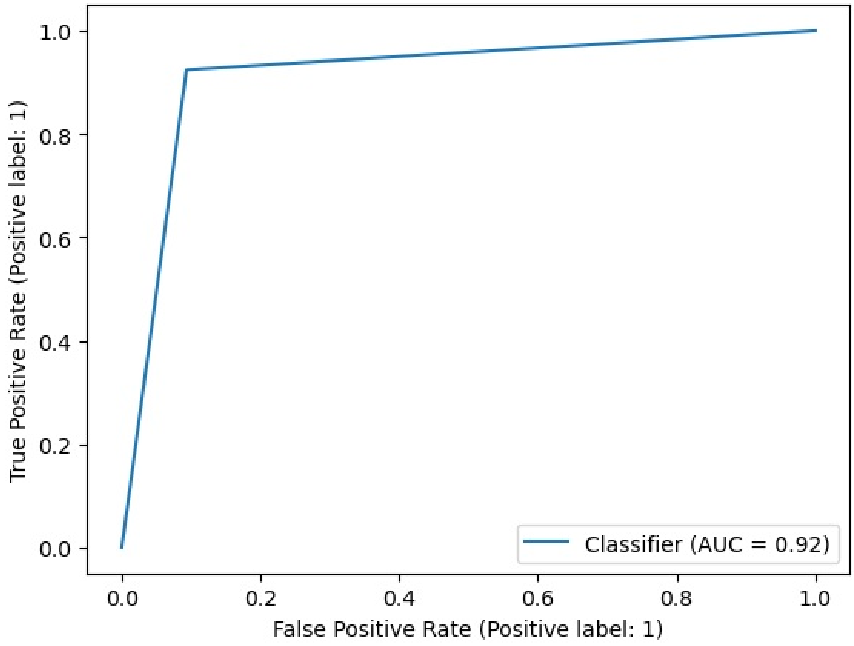


- 91.5% accuracy
- Model is well fit (train score similar to test score)
- Runs quickly (~30 seconds)
- Recall (92%) rate higher than Precision (91%)

ROC AUC Plot

In [179…
```
# display = RocCurveDisplay.from_predictions(y_test, y_pred)
```

In [180…
```
RocCurveDisplay.plot(display);
plt.savefig('./images/6-rocauc.png', bbox_inches='tight', pad_inches=0.1, facecolor='white', transparent=False)
```



Top 10 Features by Coefficients

In [181…
```
feature_names = list(final_pipe['prep'].named_transformers_['num'].get_feature_names_out()) + \
                list(final_pipe['prep'].named_transformers_['text']['tfidf'].get_feature_names_out())
```

In [182…
```
final_feats = pd.DataFrame(final_pipe['lr'].coef_).T
```

```python
final_feats.index = feature_names
final_feats.columns = ['weights']
final_feats = final_feats.reset_index()
```

In [183…

```python
final_feats_top10 = final_feats.sort_values(by='weights', ascending=False)[:10]
final_feats_top10['weights'] = final_feats_top10['weights'].round(1)
final_feats_top10.columns = ['features', 'weights']
```

In [184…

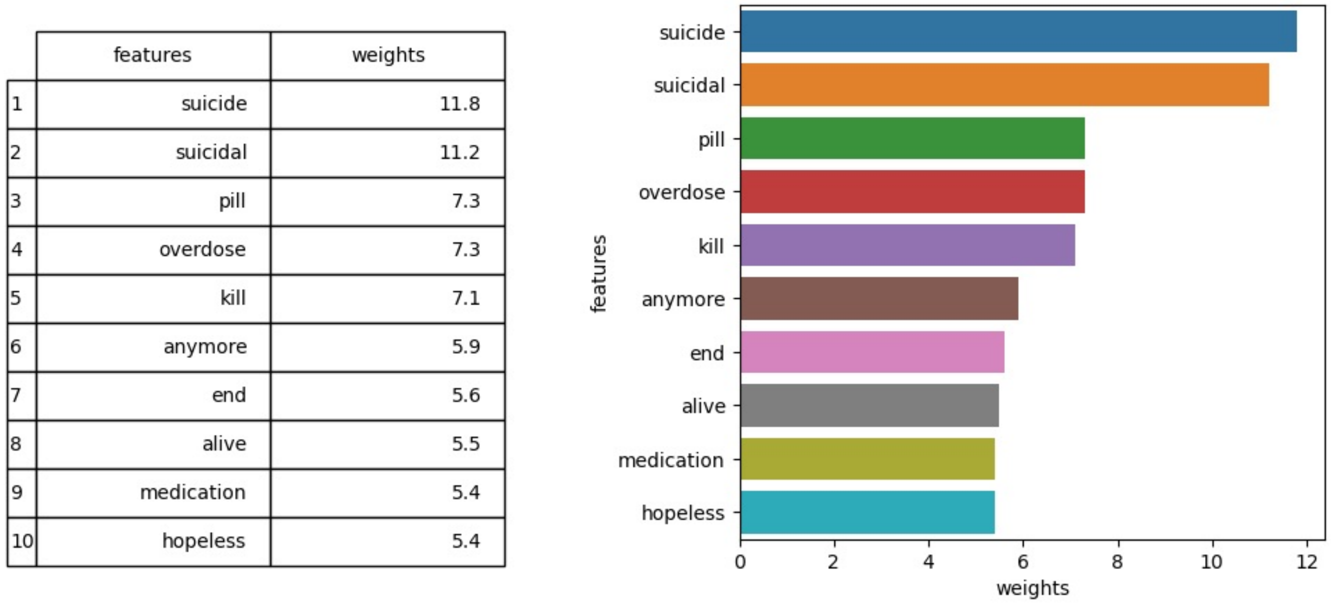
```python
final_feats_top10
```

Out[184]:

	features	weights
632	suicide	11.80
631	suicidal	11.20
499	pill	7.30
477	overdose	7.30
363	kill	7.10
46	anymore	5.90
207	end	5.60
29	alive	5.50
412	medication	5.40
334	hopeless	5.40

In [185…

```python
fig, ax = plt.subplots(1, 2, figsize=(12,5))

ax[0].axis("off");
ax[0].table(cellText=final_feats_top10.values, rowLabels=range(1,11), bbox=[0,-0.05,0.8,1], colLabels=final_fea

sns.barplot(final_feats_top10, x = 'weights', y = 'features', orient='h', ax=ax[1]);
```



	features	weights
1	suicide	11.8
2	suicidal	11.2
3	pill	7.3
4	overdose	7.3
5	kill	7.1
6	anymore	5.9
7	end	5.6
8	alive	5.5
9	medication	5.4
10	hopeless	5.4

- Top features are all terms related to suicide
- Interestingly, emotion and sentiment components are not among the top 10 features
  - Could be that emotion and sentiment are not as specific to suicidal text as these particular words
  - Likely due to multicollinearity of Emotion & Sentiment scores-- next step is to try to address this

V. Model Deployment

- This model is intended to work alongside a mental health chatbot.
- I plan to work with my stakeholders to build this model into the current framework of their mental health chatbots, in collaboration with their IT support team and clinicians to ensure the most ethical practices are used.

- So that you can see how this model would work, I have created a demo page where you can see how the model would classify user-inputted text.
  - Currently, still some issues with server deployment due to package dependencies
- For now, the model is available to test by downloading the environment yaml on Github and running the following code through

terminal:

streamlit run streamlit_app.py

Below are screenshots of the model's classification of Suicide Risk vs. Non-Risk Text:

Suicide Risk:

No Risk:

VI. Conclusions

Final Model Evaluation

The final model meets all of the project goals stated in the Introduction:

Objective #1: Accurately classify text as indicative of suicide risk vs. non risk

- 91.5% Accuracy - The model correctly classified text as indicative of suicide-risk vs. non-risk for *91.5%* of cases.

Objective #2: Minimize False Negatives * 92.5% Sensitivity (Recall) - The model minimizes *false negative classifications* * Only fails to detect suicide risk in 7.5% of cases * This is impressive, given that even [clinician recall rates](https://onlinelibrary.wiley.com/doi/10.1111/sltb.12395) range from 20 - 50%.
Objective #3: Deployed Model is able to quickly generate new predictions * Deployed Model provides predictions from new text within seconds

---

Next Steps

1. Integrate model to work alongside existing mental health chatbot frameworks

- Collaborate with clinicians to determine the best way to implement this model: what should be the response if someone is flagged for suicide risk?
  - Possible options include:
    1. Model automatically alerts on-call clinical risk team
    2. If suicide risk detected, train mental health chatbot to conduct standardized suicide risk assessment
    3. Train mental health chatbot to conduct risk-reduction techniques, such as making a safety plan

2. Continue Training Model with new data

- To improve model performance for your clients' specific needs, the model should be trained on actual chatbot conversations. The training data can be anonymized to protect client confidentiality.

3. Train Model to interpret Misspellings, Slang Words, and common Medical Abbreviations

- Current Model does not use SpellCheck because of the time/computational power required, and SpellCheck changes altering sentence meanings
  - Could help to use a spellcheck model like contextualSpellCheck that is trained not to autocorrect common slang terms and medical terms (e.g. medication names, therapy names)
- Current Model does not assign emotion/sentiment scores to Slang Words/Medical Terms
  - Update feature extraction tools to understand sentiment of these terms

4. Incorporate other Natural Language Processing Models and Techniques

- Named Entity Recognition can be used to classify terms related to top features in current model
  - e,g,: "Pill" is a top term-- associated with proper medication names (e.g. "sertraline", "klonopin", etc.)
- Using MentalBERT: pre-trained masked language models specific to mental health
- Topic Modeling / Finding semantic vectors with Genism

Loading [MathJax]/extensions/Safe.js

In [95]:
```
len(orderedcols) == len(df.columns)
```
Out[95]:
```
True
```
In [96]:
```
df = df[orderedcols]
```
In [97]:
```
df
```
Out[97]:

	target	text	neg	neu	pos	compound	anger	disgust	fear	sadness	anticipation	joy	surprise	trust
0	1	ex wife threaten suicide recently leave wife g...	0.19	0.73	0.07	-0.95	8.00	4.00	8.00	6.00	7.00	5.00	5.00	5.00
1	0	weird get affect compliment come someone know ...	0.04	0.76	0.21	0.72	0.00	1.00	0.00	0.00	2.00	1.00	1.00	2.00
2	0	finally almost never hear bad year ever swear ...	0.26	0.67	0.08	-0.70	2.00	2.00	2.00	1.00	2.00	2.00	1.00	3.00
3	1	need help help cry hard	0.29	0.40	0.31	0.11	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
4	1	lose hello name adam struggle year afraid past...	0.19	0.73	0.08	-1.00	18.00	10.00	27.00	26.00	20.00	6.00	5.00	9.00
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
232069	0	like rock go get anything go	0.08	0.92	0.00	-0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
232070	0	tell many friend lonely everything deprive pre...	0.09	0.77	0.15	0.34	1.00	1.00	1.00	1.00	0.00	0.00	0.00	1.00
232071	0	pee probably taste like salty tea someone drin...	0.00	0.84	0.16	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
232072	1	usual stuff find post sympathy pity know far b...	0.18	0.72	0.10	-0.99	9.00	6.00	7.00	10.00	3.00	1.00	1.00	3.00
232073	0	still beat first bos hollow knight fight time ...	0.26	0.69	0.05	-0.86	2.00	1.00	3.00	3.00	0.00	0.00	0.00	0.00

230072 rows × 14 columns

In [99]:
```
del df2
```

---

5. Convert to CSV/Pickle

In [100…
```
df.to_csv('../fulldataclean.csv')
```
In [101…
```
df.to_pickle('../fulldataclean.pkl')
```
In [102…
```
df.to_pickle('./data/fulldataclean.tar.gz', compression='infer')
```

Project Continued in Notebook #2

Notebook #2: 2-Modeling-and-Conclusions.ipynb

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js