Humorous content detection in user reviews

Capstone III Project

Aibek Uraimov

1 The Problem Statement

Detect humorous content in user reviews of businesses, using Yelp dataset on restaurant reviews, with at least 15,000 observations, for training a prediction model with accuracy score above 65%, and f1-score of at least 0.70 for both classes.

1.1 Context

I always wondered what makes a joke funny. Obviously, a context (as in the heading of this section!). It would be nice to have a machine be aware of and 'understand' context and then crack jokes all day. However, that is a goal for another project – in this one, I am interested in testing a basic machine learning tool to find humorous text bits online.

Yelp platform has aggregated a massive database of customer reviews on businesses for a wide range of categories. My particular interest lies in detecting funny element in customer reviews.

1.2 Scope and Goals

How to detect funny content in the sea of texts?

One approach would be to analyze reviews in conjunction with Yelp reviews' metadata: business type, geography, review user profile ratings, etc. and come up with an ensemble of linear regression and classification models. The scope of this project though would just be a classification model to rely on just the content of reviews – limited to restaurants.

Why just restaurants? Here is an initial theory – restaurants are most widely represented in Yelp and probably have the most polarized reviews and consequently, be rich in humorous remarks.

Another hypothesis I would like to check is whether there is a correlation between the rating of a business and frequency of funny content in the reviews.

The goal is to train classification models on a sampled Yelp dataset and feed it with arbitrary review texts for it to detect funniness.

1.3 Criteria for success

The criterion for success is for at least one of the models to reach a prediction accuracy score above 65% and F1 score of at least 0.70 for both classes ('Funny' and 'Not Funny').

2 Data

2.1 Key data sources

I have obtained the Open Dataset directly from Yelp's website:

- Yelp Open Dataset Link: https://www.yelp.com/dataset

The downloaded dataset included the following JSON files:

- Business - 121MB JSON file
- Check-ins - 389MB JSON file
- Reviews - 6,774MB JSON file
- Tip - 225MB JSON file
- User - 3,598MB JSON file

My target datasets were *Business* and *Reviews*.

Within these datasets, the columns of interest were:

- 'categories' - to filter on 'Restaurants' only
- 'starts' – business rating
- 'rating' – review rating
- 'review_count' – measure of a public interest
- 'funny' – the most important column, massive "labeling" for supervised learning

2.2 Constraints

Humor is subjective and what might be labelled 'funny' by one person, may not be considered so at all by another – hence, we might be getting some True Positives in a review instance, which are actually False Positives if you ask an alternative opinion. This might be a challenge in terms of integrity of the Yelp labelling.

Much of hilarious written content that goes viral does so because there is already a known context by the masses. So, even if a text appears completely neutral, it becomes very funny, once combined with an appropriate context. Our model would not 'know' no such context, so it will struggle a lot figuring out why certain review texts are getting a lot of 'funny' labels with no apparent statistical indicators.

Finally, quantitative constraints included the availability of resources:

- size of the datasets relative to the available computing equipment – aka my laptop
- time budget I had allocated for completion of the academic project aka ASAP

2.3 Loading Yelp Data

The reviews dataset was close to 7Gb, so I had to search ways to load it to Jupyter Notebook without breaking my computer, which is what I did by following this super helpful post.

Basically, what you do is load the JSON data in manageable chunks of 1MM lines using **pandas'** *pd.read_json* and indicating the *chuncksize* parameter. I then merged the Business and Reviews datasets and saved it as '.cvs' file.

2.4 Data Wrangling

2.5 Raw Data

I have extracted the *'Restaurants'* data and had to reduce the dataset even further:

- my initial attempts of feeding it into a model resulted in 2+ hours of my computer spinning the fans and significantly ruining my excitement for learning
- I reduced the still gigantic file to the last 7 years of reviews and kept only 1MM lines

2.6 Wrangling

2.6.1 Cleaning

I dropped columns that did not add anything for the purpose, such as business_id, address, latitude, longitude, is_open, attributes, categories, hours, user_id, date:

```
df.drop(df.columns[[0,1,2,5,9,11]], axis=1, inplace=True)
```

The datasets had negligible number of missing values – I simply dropped them.

2.6.2 Categorical Labels

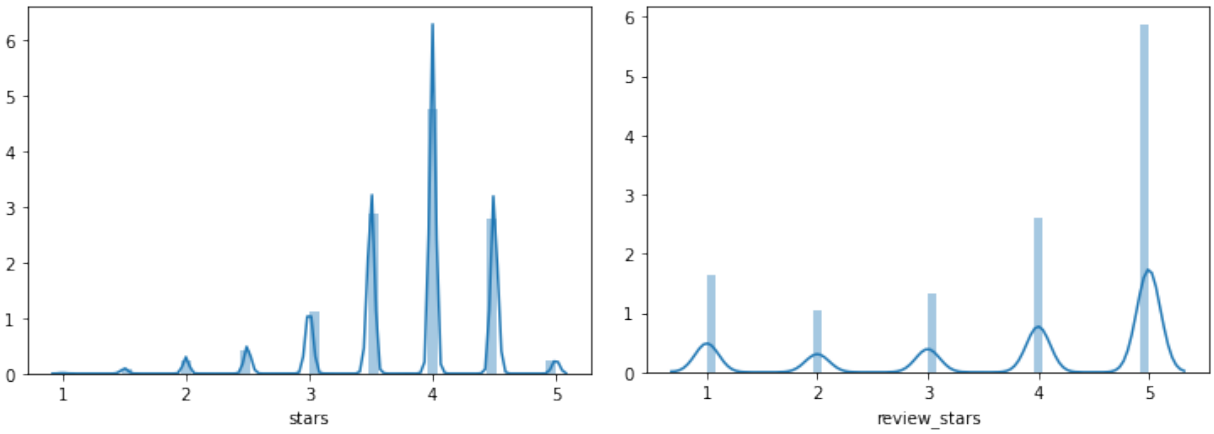
The restaurants have been given star ratings, ranging from 1 to 5 with a 0.5 increments – I reassigned them to either 'Low Rating' or 'High Rating' instead: 1 through 3.5 got to be 'Low Rating' and 4 through 5 were now 'High Rating' establishments.

The 'is funny' target variable was represented by a column of 0 or 1 values – I assigned these 'Not Funny' labels for 0s and 'Funny' for 1s.

2.7 Exploratory Data Analysis

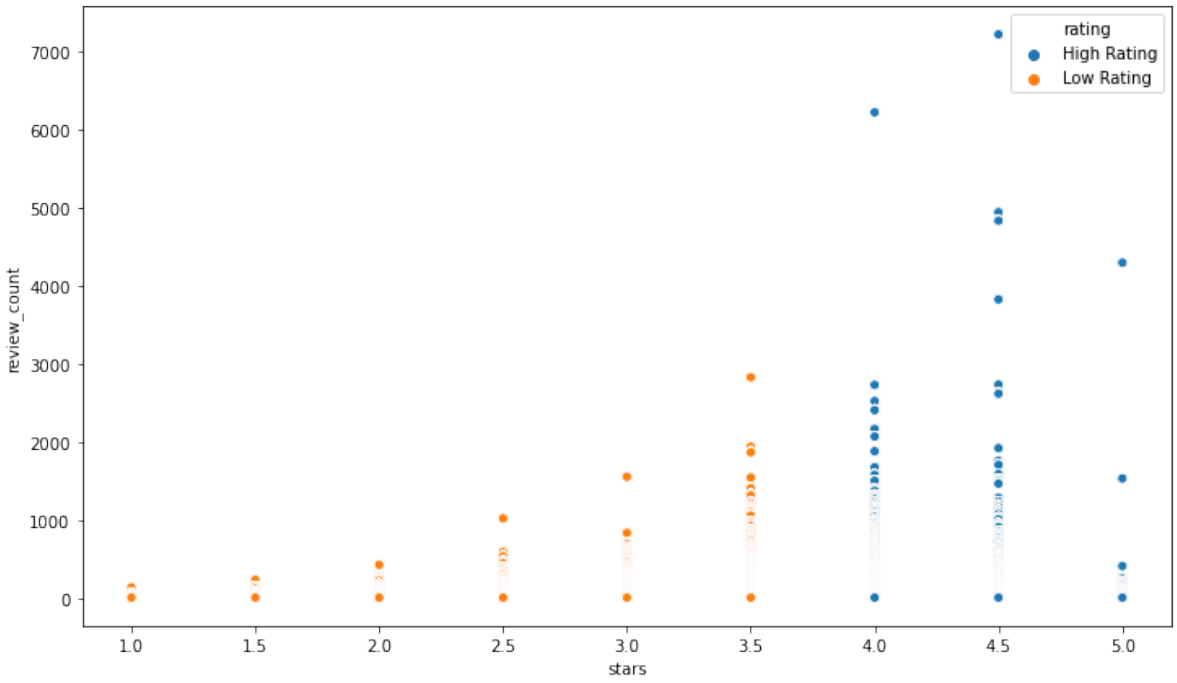
2.7.1 Distributions

Here is a distribution of Business Ratings and Review Ratings next to each other:



It looks like that most of the business ratings are within the 3.5-to-4.5-star range – a meaningful metrics, indicating that a self-regulated free-market system is in play that seems to maintain such a ratio of the thriving and not so businesses.

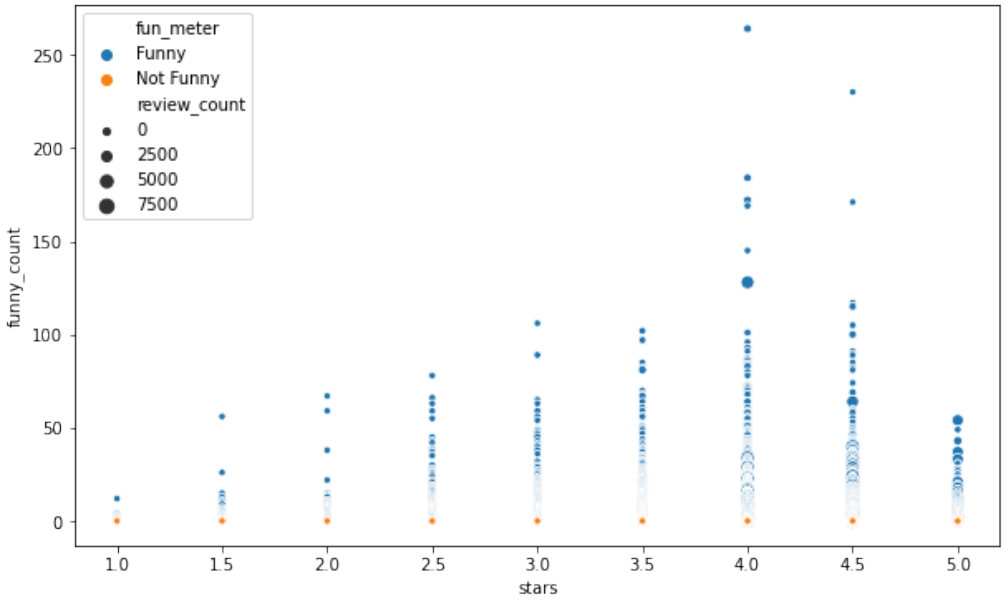
The ratings of the reviews pertain more to the writing ability of the reviewer than a business. Let's look at a distribution of review counts relative to the business ratings next:



As we can see, the users don't really bother to leave any comments for the low-star businesses – the review writers are mostly inspired with the positive experiences than the "meh" ones.

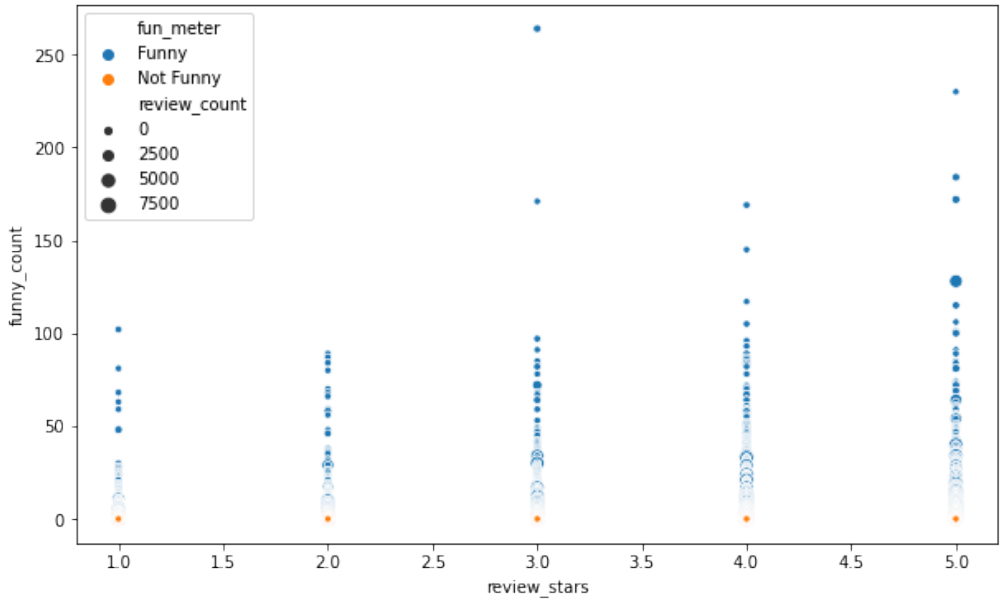
2.7.2 Correlations

Now the fun part – what are the correlations between the business ratings, review ratings and the funny meter? Let's look at restaurant stars and 'fun meter' correlation:



Well, although the count of funny reviews is consistent with the count of overall reviews, it looks like the users like to crack jokes about their bad experiences equally with the better ones.

How about ratings of reviews and 'fun meter' correlation?



Wow, will you look at that – the folks are happy to laugh at anybody's expense it looks like, be it someone rated as 'a good' reviewer or 'a bad' reviewer alike.

2.7.3 Hypothesis and Insights

Polarized Reviews

The initial theory about polarized reviews, outlined in the Scope section is confirmed: yes, high-rating restaurants attract an order of magnitude higher number of reviews than the low-rating ones.

Frequency of Funny Content

Another hypothesis was whether the frequency of funny content changes with the value of the business and review ratings. As it turns out:

- yes – on the business ratings: there are more funny reviews in the higher-rated businesses; however, that is not a reliable indicator, because it is due to higher-rated restaurants having more reviews overall
- not really – on the user review ratings: some users might not be very popular due to the probably mean things they have said; however, the content doesn't lie – human nature can't help it, if it's funny, y'all will laugh!

Insight

The insight here is that there is a huge opportunity for creating a study field of its own – what makes the reviews funny? Here's what comes to mind as elements that might be playing a role in making a given review funny:

- bad review → most likely funnier content
- businesses that involve direct interaction with customers → likely to result in more bad reviews than where there is no direct interaction, e.g., restaurants vs. digital goods
- outliers, or balance of extreme opinions → potential drama, leading to a "problematic situation"
- problematic situation is grounds for creating satire and jokes
- maybe user behavior - this is not really NLP, but it might be interesting to see correlation between circumstances of a user (e.g., all reviews are complaints) and sarcastic jokes

3 Pre-processing and training

3.1 Data Balancing

Before pre-processing work, let's check if the dataset is balanced:

```
funny_count = len(df.loc[df.fun_meter=='Funny'])

not_funny_count = len(df.loc[df.fun_meter=='Not Funny'])

print('The ratio of Funny to Not Funny in the dataset is Funny:', funny_count, ' to Not Funny: ', not_funny_count)

print('or ', round(100*funny_count/not_funny_count, 2), "% to 100%")
```

Output:

> The ratio of Funny to Not Funny in the dataset is Funny: 79672 to Not Funny: 493966
> or 16.13 % to 100%

This will lead to skewed results, so will rebalance – I split the dataset into 'Funny' and 'Not Funny' dataframes, balanced the size of the disproportionally larger 'Not Funny' dataframe to equal that of the 'Funny' one, and then concatenated them back.

The resulting dataframe now had 159,345 observations. This was still a bit large for my computer – so, I randomly dropped rows to reduce the final dataframe to 10% of the initial size:

```
df_reviews = df_set.sample(frac=0.1, replace=True, random_state=1)
```

The final dataframe to be used for modeling included the following entries and fields:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15934 entries, 128037 to 3871
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   stars         15934 non-null  float64
 1   review_count  15934 non-null  int64
 2   review_stars  15934 non-null  int64
 3   is_funny      15934 non-null  int64
 4   funny_count   15934 non-null  int64
 5   text          15934 non-null  object
 6   rating        15934 non-null  object
 7   fun_meter     15934 non-null  object
dtypes: float64(1), int64(4), object(3)
memory usage: 1.1+ MB
```

3.2 NLP Analysis with Scattertext and spaCy

Using *spaCy* open-source library with *en_core_web_sm* model, and *scattertext* tool, I set up a collection of texts from the *df_reviews* dataframe and obtained term frequency and scaled f-

score values, which were then used to create 2 dataframes – *funny_reviews* and *meh_reviews*., each with 405,272 entries. The following terms were listed with highest Funny/Not Funny scores:

- funny_reviews – top 20:

	term	Not Funny freq	Funny freq	Funny_Score	Not_Funny_Score
483	chinese	29	360	1.00	0.00
2483	chinese food	6	85	0.99	0.01
1959	airport	17	106	0.98	0.02
2006	looks like	14	103	0.98	0.02
1126	pot	32	178	0.98	0.02
2502	bagel	9	84	0.98	0.02
2983	naan	5	71	0.97	0.03
2435	website	13	86	0.97	0.03
1224	indian	35	165	0.97	0.03
2298	the drive	13	91	0.97	0.03
1958	fast food	19	106	0.97	0.03
863	lettuce	45	222	0.97	0.03
529	this location	86	335	0.96	0.04
1442	i told	36	141	0.96	0.04
609	the owner	67	301	0.96	0.04
3235	mcdonald 's	4	66	0.96	0.04
862	milk	51	222	0.96	0.04
3092	mcdonald	4	68	0.96	0.04
1031	delivery	44	190	0.96	0.04
1708	sub	30	120	0.96	0.04

- meh_reviews – top 5:

	term	Not Funny freq	Funny freq	Funny_Score	Not_Funny_Score
667	books	181	39	0.00	1.00
1419	great atmosphere	91	22	0.00	1.00
935	lobster roll	135	42	0.01	0.99
1578	screen door	83	28	0.01	0.99
1236	oyster	103	37	0.01	0.99

I will use the frequency term table to construct two biased examples to test the performance of my first model – one for 'Funny' and another for 'Not Funny' test input.

3.3 Split data into train and validation partitions

I split the data into 75/25 train/test partitions:

```
X_train, x_test, y_train, y_test = train_test_split(df_reviews.text, df_reviews.fun_meter, train_size=0.75,
random_state=42)
```

3.4 Modeling

3.4.1 Model 1 – Linear SVM Classifier

The first model I used was Support Vector Machine, a linear model for classification problems.

Using CountVectorizer() function, I created an array of words per each sentence:

```
v = CountVectorizer()

x_train_vector = v.fit_transform(X_train)

x_test_vector = v.transform(x_test)
```

After obtaining numerical data I fed the train data into the linear SVM model:

```
clf_svm = svm.SVC(kernel = 'linear')

clf_svm.fit(x_train_vector, y_train)
```

I then evaluated performance of the model on the test data:

Accuracy score:

```
clf_svm.score(x_test_vector, y_test)
```

```
0.7105923694779116
```

And f1-score:

```
f1_score(y_test, clf_svm.predict(x_test_vector), average=None)
```

```
array([0.69745474, 0.72263652])
```

This is pretty close to the objective but needs improvement – I will try more models. In the meantime, let's feed two biased examples into this model, using the frequency term dataframes.

Here is what Model 1 predicted when I fed it a review text input, biased with words from the 'funny_reviews top 20':

```
my_review = ["The place is truly a tourist trap dump! mochi doughnuts do that security ham like baby sound
like like Like what jay dim sum offerings words of wisdom"]
```

```
my_review_vector = v.transform(my_review)

clf_svm.predict(my_review_vector)[0]
```

> Output: 'Funny'

Good, that worked! OK, now a text input biased with words from the 'meh_reviews top 5':

```
not_funny_review = ["This restaurant is not very good. Giving this 3.5 stars and not coming back."]

my_review_vector = v.transform(not_funny_review)

clf_svm.predict(my_review_vector)[0]
```

> Output: 'Not Funny'

Great, this one worked too. Let's build the other 3 models.

3.4.2 Model 2 – Random Forest Classifier (Version A)

The second model I used was Random Forest Classifier, with the following parameters passed in: *bootstrap=False*, *criterion='entropy'*, and *n_estimators=100*, will label it as Version A.

I built a pipeline to vectorize the input data and then feed it into the model:

```
classifier_rf = Pipeline([('tfidf',TfidfVectorizer()),
                          ('clf',RandomForestClassifier(bootstrap= False, criterion= 'entropy', n_estimators= 100))])
classifier_rf.fit(X_train, y_train)
```

After predicting:

```
y_pred = classifier_rf.predict(x_test)

yt_pred = classifier_rf.predict(X_train)
```

the performance evaluation of Model 2 was as follows:

Accuracy score:

```
classifier_rf.score(x_test, y_test)
```

> 0.7201305220883534

f1-score:

```
f1_score(y_test, classifier_rf.predict(x_test), average=None)
```

> array([0.72325639, 0.71693323])

There is an improvement in performance compared to the Model 1 – it will be interesting to see how the models will rate the arbitrary examples – I will go over this in the Modeling Results and Analysis section further down.

3.4.3 Model 3 – Random Forest Classifier (Version B)

The third model was another Random Forest Classifier – this time with default parameters but passing the following parameters into the tfidf vectorizer in the pipeline: *stop_words='english'*, *ngram_range=(1,1)*. I will label Model 3 as Version B.

I repeated the same steps as for Model 2:

Pipeline:

```
rfc = RandomForestClassifier()

vect = TfidfVectorizer(stop_words='english', ngram_range=(1,1))

pipe_rfc = Pipeline([('vect', vect), ('rfc', rfc)])

pipe_rfc.fit(X_train, y_train)
```

Predict:

```
y_pred_rfc = pipe_rfc.predict(x_test)

yt_pred_rfc = pipe_rfc.predict(X_train)
```

Evaluate performance on the test split:

Accuracy score:

```
pipe_rfc.score(x_test, y_test)
```

```
0.7238955823293173
```

f1-score:

```
f1_score(y_test, pipe_rfc.predict(x_test), average=None)
```

```
array([0.72458688, 0.72320081])
```

It looks like using default parameters for the random forest classifier and adding stop words in the vectorizer resulted in further improvement of the model performance.

3.4.4 Model 4 – Linear SGD Classifier

The fourth and final model was a linear classifier with stochastic gradient descent (SGD) learning. Yep, I know – the thing does sound pretty cool.

Taking the same steps as before, I start with a pipeline:

```
sgdc = SGDClassifier()

vect = TfidfVectorizer(stop_words='english', ngram_range=(1,1))
```

```
pipe_sgd = Pipeline([('vect', vect), ('clf', sgdc)])
```

Predict:

```
y_pred_sgd = pipe_sgd.predict(x_test)

yt_pred_sgd = pipe_sgd.predict(X_train)
```

Evaluate performance of linear SGD classifier on the test set:

Accuracy score:

```
pipe_sgd.score(x_test, y_test)
```

```
0.7377008032128514
```

f1-score:

```
f1_score(y_test, pipe_sgd.predict(x_test), average=None)
```

```
array([0.73644388, 0.73894579])
```

Alright, it looks like Model 4 is promising to be the best one among the four I have prepared so far.

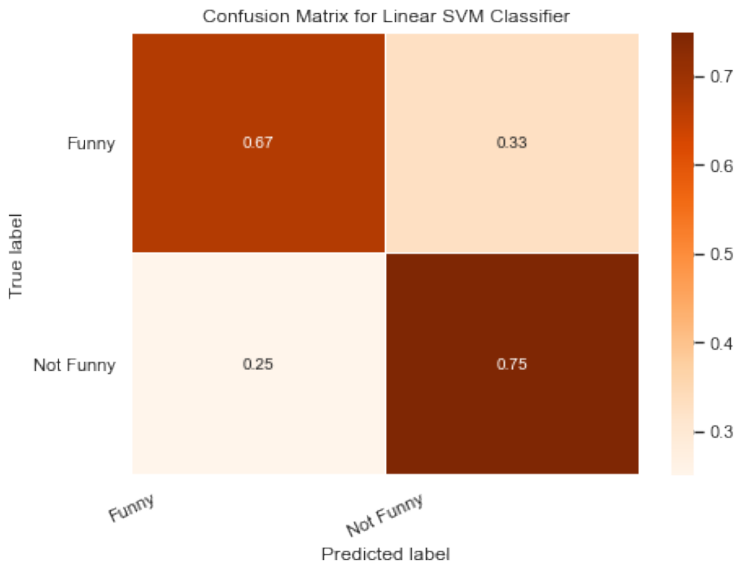
3.5 Confusion Matrices

In addition to calculating accuracy and f1-scores, I have also prepared Confusion Matrices for each of the models. Here they are:

3.5.1 Model 1 – Linear SVM Classifier Confusion Matrix

Here are the Model 1 results:

```
Confusion Matrix :
 [[1329  651]
 [ 502 1502]]

Test Set Accuracy Score :
 0.7105923694779116

Train Set Accuracy Score :
 0.9834309623430962

Classification Report :
              precision    recall  f1-score   support

       Funny       0.73      0.67      0.70      1980
   Not Funny       0.70      0.75      0.72      2004

    accuracy                           0.71      3984
   macro avg       0.71      0.71      0.71      3984
weighted avg       0.71      0.71      0.71      3984
```



3.5.2 Model 2 – Random Forest Classifier (Version A) Confusion Matrix

Model 2 results:

Confusion Matrix :

```
 [[1457  523]
 [ 592 1412]]
```

Test Set Accuracy Score :

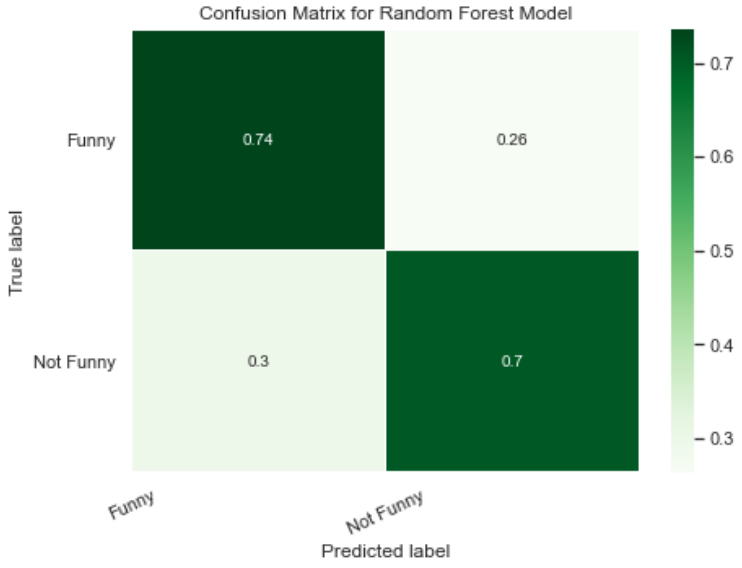
```
 0.7201305220883534
```

Train Set Accuracy Score :

```
 1.0
```

Classification Report :

	precision	recall	f1-score	support
Funny	0.71	0.74	0.72	1980
Not Funny	0.73	0.70	0.72	2004
accuracy			0.72	3984
macro avg	0.72	0.72	0.72	3984
weighted avg	0.72	0.72	0.72	3984



3.5.3 Model 3 - Random Forest Classifier (Version B) Confusion Matrix

Model 3 results:

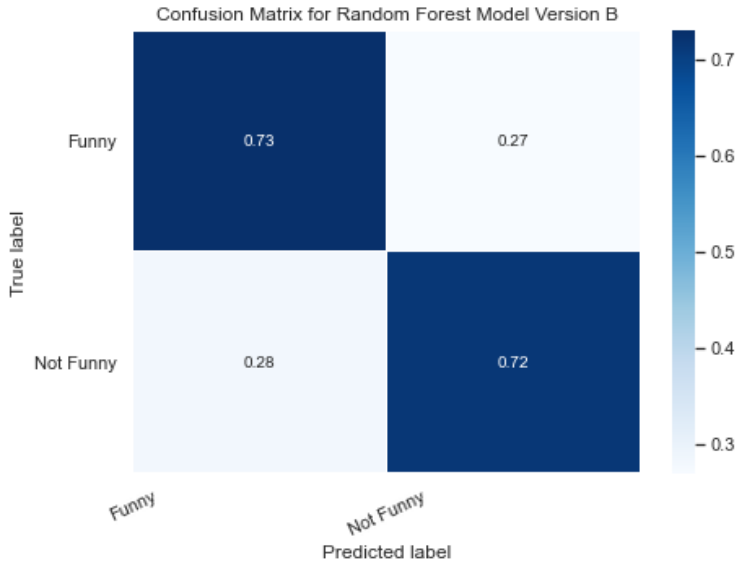
Confusion Matrix :
```
 [[1447  533]
 [ 567 1437]]
```

Test Set Accuracy Score :
```
 0.7238955823293173
```

Train Set Accuracy Score :
```
 1.0
```

Classification Report :

	precision	recall	f1-score	support
Funny	0.72	0.73	0.72	1980
Not Funny	0.73	0.72	0.72	2004
accuracy			0.72	3984
macro avg	0.72	0.72	0.72	3984
weighted avg	0.72	0.72	0.72	3984



3.5.4 Model 4 - Linear SGD Classifier Confusion Matrix

Model 4 results:

Confusion Matrix :
```
 [[1460  520]
 [ 525 1479]]
```

Test Set Accuracy Score :
```
 0.7377008032128514
```

Train Set Accuracy Score :
```
 0.8638493723849372
```

Classification Report :

	precision	recall	f1-score	support
Funny	0.74	0.74	0.74	1980
Not Funny	0.74	0.74	0.74	2004

```
              precision    recall  f1-score   support

    accuracy                         0.74      3984
   macro avg       0.74      0.74      0.74      3984
weighted avg       0.74      0.74      0.74      3984
```

3.6 Model Scenarios

If only there was a way to pit them against each other on an equal basis… Wait, there is!

I came up with four arbitrary text input examples to be used for testing each of the four models. Here they are:

- Example 1:

```
my_review_1 = ["The place is truly a touristic trap dump! It looks like a restaurant, but it is not!"]
```

- Example 2:

```
my_review_2 = ["I told the owner that mochi and doughnuts were speaking to me the words of wisdom"]
```

- Example 3:

```
my_review_3 = ["Can't even explain how much I love this place"]
```

- Example 4:

```
my_review_4 = ["This restaurant is not very good. Giving this 3.5 stars and not coming back."]
```

Now that the examples ready, let's test this thing on all four models.

4 Modeling Results and Analysis

Here are the results of predictions made on the four uniform examples of input review texts, as determined by each of the four models.

4.1 Model 1 - Linear SVM Classifier

Using the four review text input examples, prepared earlier, let's predict on them with Model 1:

4.1.1 Example 1 – Input and Output

```
my_review_1 = ["The place is truly a touristic trap dump! It looks like a restaurant, but it is not!"]

my_review_vector_1 = v.transform(my_review_1)

clf_svm.predict(my_review_vector_1)[0]
```

```
'Funny'
```

4.1.2 Example 2 – Input and Output

```
my_review_2 = ["I told the owner that mochi and doughnuts were speaking to me the words of wisdom"]

my_review_vector_2 = v.transform(my_review_2)

clf_svm.predict(my_review_vector_2)[0]
```

```
'Not Funny'
```

4.1.3 Example 3 – Input and Output

```
my_review_3 = ["Can't even explain how much I love this place"]

my_review_vector_3 = v.transform(my_review_3)

clf_svm.predict(my_review_vector_3)[0]
```

```
'Not Funny'
```

4.1.4 Example 4 – Input and Output

```
my_review_4 = ["This restaurant is not very good. Giving this 3.5 stars and not coming back."]

my_review_vector_4 = v.transform(my_review_4)

clf_svm.predict(my_review_vector_4)[0]
```

```
'Not Funny'
```

4.2 Model 2 - Random Forest Model (version A)

4.2.1 Example 1 – Input and Output

```
my_review_1 = ["The place is truly a touristic trap dump! It looks like a restaurant, but it is not!"]

classifier_rf.predict(my_review_1)[0]
```

```
'Not Funny'
```

4.2.2 Example 2 – Input and Output

```
my_review_2 = ["I told the owner that mochi and doughnuts were speaking to me the words of wisdom"]

classifier_rf.predict(my_review_2)[0]
```

```
'Funny'
```

4.2.3 Example 3 – Input and Output

```
my_review_3 = ["Can't even explain how much I love this place"]

classifier_rf.predict(my_review_3)[0]
```

```
'Not Funny'
```

4.2.4 Example 4 – Input and Output

```
my_review_4 = ["This restaurant is not very good. Giving this 3.5 stars and not coming back."]

classifier_rf.predict(my_review_4)[0]
```

```
'Not Funny'
```

4.3 Model 3 - Random Forest Model (version B)

4.3.1 Example 1 – Input and Output

```
my_review_1 = ["The place is truly a touristic trap dump! It looks like a restaurant, but it is not!"]

pipe_rfc.predict(my_review_1)[0]
```

```
'Not Funny'
```

4.3.2 Example 2 – Input and Output

```
my_review_2 = ["I told the owner that mochi and doughnuts were speaking to me the words of wisdom"]

pipe_rfc.predict(my_review_2)[0]
```

```
'Funny'
```

4.3.3 Example 3 – Input and Output

```
my_review_3 = ["Can't even explain how much I love this place"]

pipe_rfc.predict(my_review_3)[0]
```

```
'Not Funny'
```

4.3.4 Example 4 – Input and Output

```
my_review_4 = ["This restaurant is not very good. Giving this 3.5 stars and not coming back."]

pipe_rfc.predict(my_review_4)[0]
```

```
'Not Funny'
```

4.4 Model 4 – Linear Classifier with SGD

4.4.1 Example 1 – Input and Output

```
my_review_1 = ["The place is truly a touristic trap dump! It looks like a restaurant, but it is not!"]

pipe_sgd.predict(my_review_1)[0]
```

```
'Funny'
```

4.4.2 Example 2 – Input and Output

```
my_review_2 = ["I told the owner that mochi and doughnuts were speaking to me the words of wisdom"]

pipe_sgd.predict(my_review_2)[0]
```

```
'Funny'
```

4.4.3 Example 3 – Input and Output

```
my_review_3 = ["Can't even explain how much I love this place"]

pipe_sgd.predict(my_review_3)[0]
```

```
'Not Funny'
```

4.4.4 Example 4 – Input and Output

```
my_review_4 = ["This restaurant is not very good. Giving this 3.5 stars and not coming back."]

pipe_sgd.predict(my_review_4)[0]
```

```
'Not Funny'
```

4.5 Model Selection

Here is a summary of how the models have performed:

Model 1:		
	Accuracy Score:	0.7105923694779116
	f1_score:	[0.69745474, 0.72263652]
Model 2:		
	Accuracy Score:	0.7201305220883534
	f1_score:	[0.72325639, 0.71693323]
Model 3:		
	Accuracy Score:	0.7238955823293173
	f1_score:	[0.72458688, 0.72320081]
Model 4:		
	Accuracy Score:	0.7377008032128514
	f1_score:	[0.73644388, 0.73894579]

Based on the Confusion Matrix metrics, I selected Model 4 as the best model for running prediction scenarios.

5 Summary and Conclusion

5.1 Summary

Now that we have tested all four models, let's summarize the outcome of the tests:

	Input	Outcome
Model 1 - Linear SVM Classifier	Example 1	Funny
	Example 2	Not Funny
	Example 3	Not Funny
	Example 4	Not Funny
Model 2 - Random Forest Classifier (Version A)	Example 1	Not Funny
	Example 2	Not Funny
	Example 3	Not Funny
	Example 4	Not Funny
Model 3 - Random Forest Classifier (Version B)	Example 1	Funny
	Example 2	Not Funny
	Example 3	Not Funny
	Example 4	Not Funny
Model 4 - Linear SGD Classifier	Example 1	Funny
	Example 2	Funny
	Example 3	Not Funny
	Example 4	Not Funny

5.2 Conclusion

Judging by performance evaluations, we would be better off sticking with Model 4 - after all it seems to have the best sense of humor!!! Also, let us not forget that the 'funniness' of the texts is always going to be subjective - at least by us, the humans.

5.3 Recommendations (Further Work)

I am definitely going continue work in exploring ways to detect funniness around us – I think it is key to getting close to unleashing the power of true AI. I suspect it will take a slightly different approach than just applying statistics to text.

As for the potential applications stemming from this exercise, here are a few: 1 – bots for online games (quests) that require 'conversations' with the gamer, 2 – platform for detecting and aggregating written content with high chances of going viral on social media, 3 – research on building the true AI.

Here's a brain teaser as a parting offering:

- people react strongly to depictions of potentially hazardous/critical/dangerous situations – especially if funny element is present

- Why? Because funny implies a more or less harmless, or at least, bearable to watch resolution of those situations
- Combination of “situation” + “resolution” => ”funny” is basically a highly valuable lesson that humans may use, should they find themselves in a similar emulation, I mean, situation
- Valuable lessons == high interest => viral content sharing

Thank you for your attention!