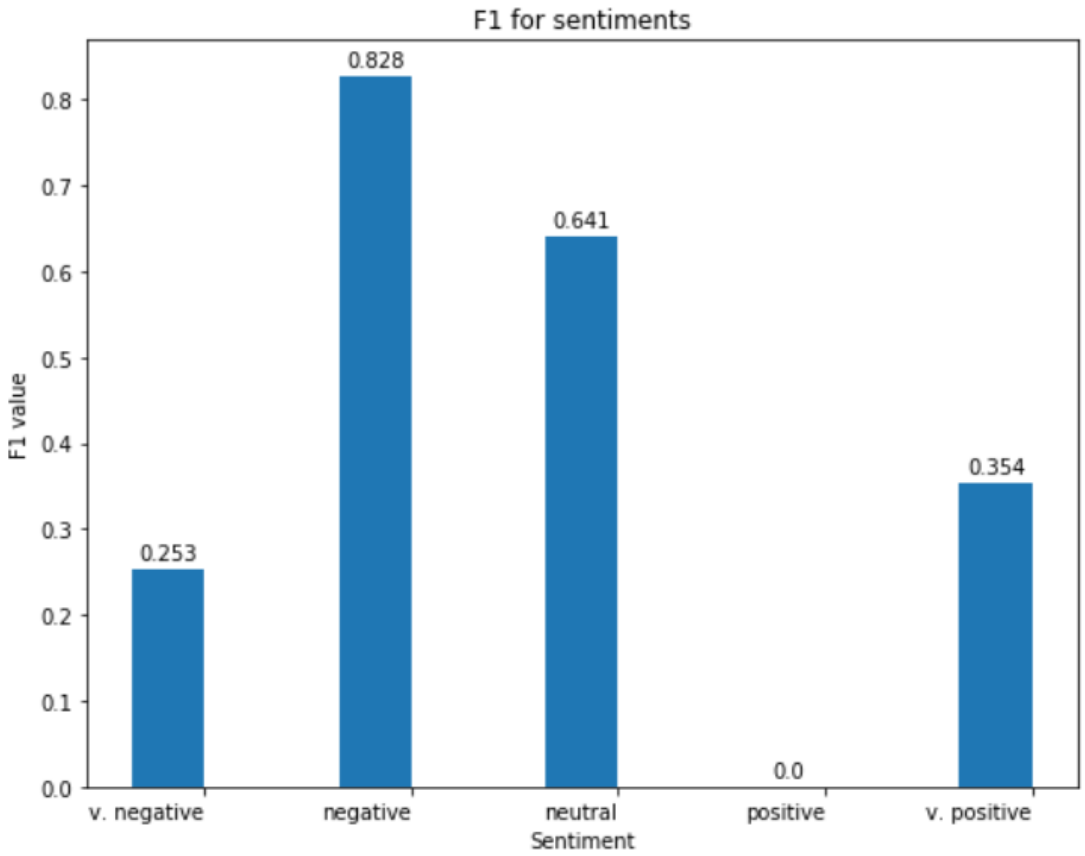
Q1

i) Results - For the above classifiers report the classifier accuracy as well as macro-averaged precision, recall, and F1 (to three decimal places). Show the overall results obtained by the classifiers on the training and test sets in one table, and highlight the best performance. For the best performing classifier (by test macro F1) Include a bar chart graph with the F1 score for each class - (sentiment polarity labels on x-axis, F1 score on Y axis)

	Precision	Recall	F1
Most frequent			
accuracy			0.631
macro avg	0.126	0.200	0.155
Stratified			
accuracy			0.478
macro avg	0.205	0.202	0.204
weighted avg	0.477	0.478	0.477
Logrec onehot			
accuracy			0.747
macro avg	0.543	0.377	0.415
	Precision	Recall	F1
Logrec tf-idf			
accuracy			0.733
macro avg	0.572	0.325	0.352
SVC onehot			
accuracy			0.698
macro avg	0.283	0.260	0.254
SVC tf-idf			
accuracy			0.637
macro avg	0.498	0.205	0.165

The validation set had the following proportions:

Sentiment	**Number thereof (validation set)**
very negative	215
negative	1961
neutral	845
positive	15
very positive	73



The dummy classifiers are just to provide baselines. Most frequent categorises every entry as the most common one, which gives very low f1 measures, and unusually competitive accuracy given the imbalanced dataset.

Stratified takes the probabilities of each category and assigns them to each point randomly.

Logistic Regression with a onehot encoding gave the best results.

This is somewhat peculiar, as onehot encoding is rather crude, being a simple Boolean vector showing the presence or absence of a word in each document.

Term Frequency Inverse Document Frequency is an encoding method which accounts for document frequency (count of the term against all in document) and inverse document frequency log(documents/documents containing the term).

Theoretically, this should filter out words that have low information like stop-words while retaining words that frequently appear in the document class.

However, the TF-IDF encodings consistently perform worse than the one-hot with this dataset.

This could be because of the extremely low numbers of positive and very positive comments making those seem even less likely, or maybe a lack of stop words in the informal writings caused important words to be excluded.

Logistic regression is based on passing sentiment weights through a logistic curve to keep them confined to probability ranges. This forms the basis of estimating the probability of being in a category. The actual core being used is Limited memory BFGS, which is more complex, but for our purposes being probabilistic is the main difference from SVC.

The SVC is based on plots of the data points and fitting curves around them. My basis for including a TF-IDF version was that it would have more useful locations as a result of its weights on more relevant words and produce a better shape for classification. The onehot version had great difficulty with the smaller classes.

It did produce better precision, so returned more correct results as a fraction (on average), but having lost recall, had found less of them (on average). The harmonic mean result for that (0.165) comes from some very poor results for everything that's not exactly 'negative'. This is a general problem with unbalanced datasets, where the classifier can become 'overfitted', that is, overrepresent the dominant data without losing closeness at the cost of predicting anything.

2

i) Classifier - Regularization C value. The c value defaults to one, where a typical value is apparently around 10^4 ± an order of magnitude. Smaller C values give less fitted results, so I started with 1000 to see if the minority data would be more noticeable.

Average f1 climbs to .479 with neglible loss in 'negative'.

10000- made no perceptible difference. The C was set back to 1000 and attention turned to the vectorizer.

Vectorizer- Sublinear_tf means that the term frequency part of term frequency against idf scales logarithmically instead of linearly. Thus there will be a less pronounced difference between words of different frequencies, and the inverse document frequency will have more effect as it defaults to log values. This will remove common words that have low information more effectively.

After enabling sublinear_tf, the macro average improved slightly to .494.

Max features limits the vocabulary. This one was already set for 20000 for time considerations, although it was only 23832 anyway.

Setting to 19k gave a macro average of .491, so we're starting to shave words that are actually common enough to be predictive.

21k gave .486, including a swathe of useless vocabulary.

My choice was stopwords=English, which trims out the known stopwords from the vocabulary, doing much the same as max_features.

Unfortunately, this lost macro average f1 down to .475, so that was rescinded.

Why that might have happened is presumably because the idea of stopwords as carrying no information is a simplification, and this dataset happened to have some that it could use to discern sentiment. Perhaps the positive ones are generally shorter, for example.

Next, attempted to adjust class weights, which changes the penalty for mistakes in the classes to not be proportional to how many there are. Thus most of them sum to 1.

Weights 1:

```
class_weights= {"very negative":0.1, "negative":0.01, "neutral":0.19, "
positive":0.4, "very positive":0.3})
```

were added, hoping to emphasise the underrepresented classes.

This decreased the macro average to .458, from the very negative being decreased.

Weights 2:

```
class_weight= {"very negative":0.2, "negative":0.01,"neutral":0.19, "positive":0.4, "very positive":0.2}
```

Weights 3

```
class_weight= {"very negative":0.25, "negative":0.01,"neutral":0.14, "positive":0.35, "very positive":0.25}
```

None	Weights 1	Weights 2	Weight 3	4	5	6
Vp- 425	202	203	215	222	214	195
neg- 806	809	810	806	806	804	809
n- 641	640	641	637	639	641	639
p- 207	261	308	286	258	111	200
Vp- 390	379	340	386	386	345	379
Macro avg- 497	458	460	466	462	453	445

Weights 4, abandoned sum to 1.

```
class_weight= {"very negative":0.8, "negative":0.01,"neutral":0.14, "positive":0.35, "very positive":0.25}
```

Weights5

```
class_weight= {"very negative":0.05, "negative":0.01,"neutral":0.14, "positive":0.5, "very positive":0.25}
```

Weights 6

```
class_weight= {"very negative":0.07, "negative":0.01,"neutral":0.19, "positive":0.4, "very positive":0.31}
```

At this point I concluded that I was not going to recover very negative's classification loss. The very negatives make up 6.9% of the data, but none of the weights whether larger or smaller come close to the base. Improving the positive classification slightly is never going to be worth it, because it's 0.048% of the data.

As to why the classifiers ability with very negatives plummets, I can say fiddling with the class weights dramatically impacts the recall (.388->~0.12ish). The fact that both increasing AND reducing the penalty for mistakes in classification dramatically hurts the proportion of true very negatives found presumably means that the classifier's proportional penalty is exactly right. But even when I'm almost on top of that value I gain no meaningful recall, which I don't know how to redress.

Finally I tried increasing the ngram range from unigrams only to unigrams and bigrams. This means the model will consider more context from word pairs, as well as greatly increase in size and running time.

As such, max_features is removed for the moment. This brings the macro avg down to .458. Re-instituting max_features puts us back up to .475. After quickly checking both sides of the features again (21000 and 19000) and seeing they both caused declines, I increased the max iterations, doubling them to 2000 assuming that bigram relationships might need longer to converge. This did nothing.

The explanation for this is most likely that the data is too sparse for bigrams to make sense, especially in the less well-populated classes. Moving to higher dimensions like bigrams needs more data to create a 'shape', which is not possible for this sample.

For the test data:

	Precision	Recall	F1
Accuracy			0.736
Macro avg	0.560	0.488	0.516

The results are better than for the validation, which shows the model has not over fitted.

2 ii) Error analysis:

Manual check on the sentiment classification.

Given that this is quite subjective, twenty of the comments were selected along with their answers and manually inspected first. None of them seemed egregiously wrong, but for example:

`Spotify is sooooo awesome!!! They even get new albums like Mumford and Sons and Green Day's new album the day they're released! It's revolutionary and the future`

Had a `positive` sentiment despite seeming in line with the `very positive` comments, but as was said, that's just subjectivity and not evidence of a problem like the answers being mismatched.

The training set consists of:

Sentiment	Number thereof (training set)
`very negative`	`97`
`negative`	`878`
`neutral`	`7679`
`positive`	`3231`
`very positive`	`253`

So the smallest is very negative at 97/12138 = 0.80%, which may benefit from upsampling.

Examining classification errors from the training set shows that the classifier does not understand emoticons or other expressive punctuation very well, as that is all of the errors there. There are not too many of them (11).

Next, examining the classifications for the validation set shows a few trends. The classifier was quite confused by a discussion on the use of fresh and homemade in advertising, presumably because those terms are overrepresented in glowing testimonials and advertising.

Posts with large amounts of text like reviews and rambling show up a lot with wrong neutral predictions, which would be because their sentiments are conveyed in a small percentage of the words, which even if you're TFIDFing the stop words seems to drag down the sentiment.

3) Majority type is an attribute covering what type of comment it is, such as an agreeable response or a critical response. This extra information should help contextualise a post, and some of the categories even reveal it directly like 'disagreement'.

But the results on the validation data showed a drop in macro average F1 down to .481. Apparently, the majority type just doesn't correlate enough with sentiment polarity to help.

Confusion matrices key:

Very neg negative neutral positive very positive

```
[[  81  125    8    1    0]
 [  79 1637  219   10   16]
 [   4  314  521    0    6]
 [   2   10    0    3    0]
 [   0   17   33    0   23]]
```

Fig 1: Confusion matrix for the setting starting.

```
[[  76  128    9    1    1]
 [  67 1644  223   13   14]
 [   3  323  513    0    6]
 [   3    9    0    3    0]
 [   0   17   35    0   21]]
```

Fig 2. Confusion matrix with majority_type.

Addressing the issues with punctuation, the next attempt involved changing the vectoriser to character n-grams instead of words.

While this did help recognise the very positive emoticons, the overall results were dismal.

The macro average F1 dropped to 0.215, as it largely lost the ability to classify anything except neutral, as can be seen from the confusion matrix.

This is almost certainly because examining the features list reveals it to only be full of character unigrams, which knowing the frequency of will not help nearly as much as words.

```
[[   2  208    3    2    0]
 [   2 1902   55    0    2]
 [   0  769   73    0    3]
 [   0   13    1    1    0]
 [   0   63    9    0    1]]
```

Fig 3. Confusion matrix for char n-grams.

Thus the vectorizer returned to words, but now with the token pattern `r'[^\s]+'`,anything that is not whitespace.

This couldn't achieve a higher macro average F1 either, coming from losses in classifying very negative and general positive.

```
[[  69  137    7    2    0]
 [  73 1644  234    2    8]
 [   5  323  510    0    7]
 [   3   10    1    1    0]
 [   0   13   39    0   21]]
```

Fig 4. Confusion matrix for punctuation in tokens.

Examining the features gives a plausible explanation for this, being saturated with tokens like '(a' and '.exe' or '---' which provide little information and dilute the informative classes.

4)

For the subreddit prediction, a pipeline was set up performing TFIDF classification on the body and thread title of our posts, and a onehot count on our author, which will hopefully lead to them being seen as a connected category, as they doubtless have subreddits they frequent.

Our baseline classifier is a logistic regression at c=10.

	Precision	Recall	F1
Accuracy			0.592
Macro avg	0.620	0.433	0.462

Table for baseline logistic regression.

Examining the individual classifications shows it has a tremendous problem with summonerschool, confusing them for posts from the actual league of legends thread. Fortunately, it is using the summonerschool classification, just exclusively on posts from LOL.

The first of the classifiers chosen to investigate was k nearest neighbour, as it is not parameter-based, so is less adversely affected if the boundaries are somewhat fuzzy.

This had a great deal of trouble with the less populated threads, so grid search optimisations began.

```
'classifier__weights': ('uniform', 'distance'),
'classifier__leaf_size': (1, 100),
'classifier__n_neighbors': (1, 10),
```

Were placed into the grid, however only n_neighbors had any effect, bringing the metrics to:

	Precision	Recall	F1
Accuracy			0.421
Macro avg	0.404	0.237	0.262

Table for K nearest-neighbours

Which is precipitously dropping from our baseline. This is likely because with sparse datasets there simply are not enough neighbours for accurate classification, and being a simple majority of neighbours it cannot distinguish threads by keywords as easily as logrec, as that predicts the probability of belonging to a group based on linear trajectory.

So next, the various support vector machines.

```
'classifier__kernel': ('linear', 'poly','rbf','sigmoid','precomputed'
'classifier__C': (0.01,0.1,1,10,100,1000),
```

```
[Parallel(n_jobs=1)]: Done  60 out of  60 | elapsed: 48.2min finished
Best score: 0.381
Best parameters set:
        classifier__C: 10
        classifier__kernel: 'sigmoid'
```

After a very long grid search, we received our parameters.

	Precision	Recall	F1
Accuracy			0.551
Macro avg	0.521	0. 367	0. 384

Even with optimal parameters, it remains below the logistic regression. Given this and the neighbours it seems reasonable to conclude that this dataset is not friendly to geometric methods.

Hence the last classifier will be Adaboost. This is based on applying a series of weak (simple & not very representative) models to the training data, and then applying them again but adjusting the weights to focus on the incorrectly classified instances. Then they vote on what to classify as.

```
'classifier__base_estimator': (None,loglass),
'classifier__n_estimators': (50,100,200,500),
```

The base estimators are the models we are using. None is actually the default, decision trees with maximum depth 1. The other is a logistic regression identical to our base.

The n estimators are how many estimators we use before leaving.

The logistic regresion with 500 iterations was the best.
Despite that, and taking 15 minutes to run, the results were entirely unsatisfactory.

	Precision	Recall	F1
Accuracy			0.478
Macro avg	0. 522	0. 202	0.212

Table for Adaboost

The recall of this classifier was dismal, with many entire classes going without a single correct classification.
The reason that many logistic regressions can be worse than a single is probably because of adaboost's focus on unclassified instances. Some of these may just not be representative, so making a bunch of models to find them will not help.

Being a weighted vote also makes it more vulnerable to unbalanced data sets like this one, as the models that know those will remain a minority.