COMP9417

Machine Learning and Data Mining

Text Classification Algorithm

2020 - Term 1

COMP9417 Group Project T1 2020

Riyaz Ahmed, Mohamed Rabeek	Z5258129
Thazmeel Ahmed, Mohamed Rabeek	Z5209070
Thomas Horspool	Z5019744
Noah Lackey,	Z5169776
Prem Pant,	Z5259743

1. Introduction

This report will show the steps the Parasite team used to offer conclusions and provide a detailed explanation and justification of methods used, while at times summarizing, reflecting and drawing conclusions from the information presented in this report such as Precision, Recall and F1.

The purpose of this report stems from the need to present a solution to the problem of recommending users find the most interesting articles according to their preferred topics. This project aims to train, tweak and perfect text classification machine learning models and use the model to predict the most relevant news articles for each of the 10 users, that are interested in one of 10 topics: ARTS CULTURE ENTERTAINMENT, BIOGRAPHIES PERSONALITIES PEOPLE, DEFENCE, DOMESTIC MARKETS, FOREX MARKETS, HEALTH, MONEY MARKETS, SCIENCE AND TECHNOLOGY, SHARE LISTINGS, and SPORTS.

One main issue that needs to be addressed is the datasets feature size. The number of features is much bigger than the number of instances. Some pre-processing using the vectorizer of choice is needed to be done in order to reduce the dimensionality of input and not worsen the accuracy. Another issue that needs to be addressed is filtering out only the relevant words to be the input features and finding a way to remove the misspelled words from the feature space. One technique that can be used is to set some hyperparameters that remove words from the feature space if they don't at least appear a certain number of times.

2. Exploratory data analysis

`data_analysis.py` contains several functions that we used to obtain the results shown below. The data is loaded in, with a list of lists of included words, and a list of correct classifications.

```
import pandas as pd
Import numpy as np

data = pd.read_csv('training.csv')
word_lists = np.array(data.iloc[:,1])
y = np.array(data.iloc[:,2])
```

The frequency of each class is an important metric. The `class_count` function iterates the list of correct classifications, and counts each occurence.

```
def class_count():
    d = {}
    for c in y:
        d[c] = d.get(c, 0) + 1
```

The table below displays the counts, the total, and a rounded percentage for each class. As expected, roughly 50% of the articles are irrelevant. More interesting is the extremely low number of science articles, along with a similarly low count for arts and domestic markets, as well as less alarming counts for a few other categories.

Figure 1: Pie chart demonstrating the percentage split of classes within the dataset

Count distribution for each of the classes

Class	Count
Arts/culture/entertainment	117
Biographies/personalities/people	167
Defence	258
Domestic markets	133
Forex markets	845
Health	183
Irrelevant	4734
Money markets	1673
Science and technology	70
Share listings	218
Sports	1102
TOTAL	**9500**

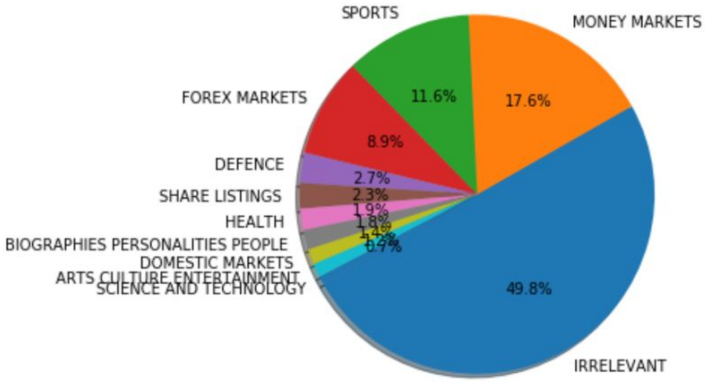


Table generated from code found in data_analysis.py

Words that appear in titles either very frequently or very infrequently provide little classifying information.

3. Methodology

Before we could select the models that we would continue hyper-parameter tuning on, we ran a simple benchmark for a range of prospective models, and two types of text vectorisation. The code used is found in `benchmark.py`.

We decided to use the `StratifiedKFold` object, as it provides splits with equal classification distribution. This best accounts for the behaviour explained in the project specification; the class distribution of the provided training set will resemble the distribution of the hidden test set and represt the skew towards our 'irrelevant' class at the same time. The default number of splits is 5, which we did not alter, for the purposes of a balance between lower computation time and higher evaluation accuracy.

The output was formatted and averages calculated, to be displayed in the table below. The bold values represent the average over the 5 folds for each model, under each vectorisation.

Figure 2: Mean accuracy through five-fold cross validation performed on each model with both binary and normalised vectorizer

		Random Forest	Multinomial NB	Linear SVC	Decision Tree	Dummy
Binary	1	0.7252631579	0.72	0.7226315789	0.6815789474	0.4984210526
	2	0.7236842105	0.7247368421	0.7294736842	0.6936842105	0.4984210526
	3	0.7168421053	0.7189473684	0.7284210526	0.6621052632	0.4984210526
	4	0.7221052632	0.7278947368	0.7210526316	0.6873684211	0.4984210526

	5	0.7284210526	0.7252631579	0.7273684211	0.6757894737	0.4984210526
	Av.	**0.7232631579**	**0.7233684211**	**0.7257894737**	**0.6801052632**	**0.4984210526**
Normalised Count	1	0.7315789474	0.6463157895	0.7663157895	0.7010526316	0.4984210526
	2	0.7342105263	0.6452631579	0.7789473684	0.6936842105	0.4984210526
	3	0.7326315789	0.65	0.7768421053	0.6768421053	0.4984210526
	4	0.7257894737	0.6531578947	0.7647368421	0.6847368421	0.4984210526
	5	0.7336842105	0.6452631579	0.7652631579	0.6778947368	0.4984210526
	Av.	**0.7315789474**	**0.648**	**0.7704210526**	**0.6868421053**	**0.4984210526**

The code for this file can be found in benchmark.py

As seen from our initial benchmarking (figure 2 above), the Linear SVC with a normalised word count feature vectorisation achieved the highest accuracy score at roughly 0.77. A broader range of metrics were also considered to obtain better comprehension of our data. The metrics include precision, f1 and recall.

Figure 3: Precision score from five-fold cross validation performed on each model with Normalised Vectorizer (this is the vectorizer we prefer based on the prediction of previous tables and its additional functionality)

		Random Forest	Multinomial NB	Linear SVC	Decision Tree	Dummy
Normalised Count	1	0.6144094860161302,	0.29659926370340517,	0.6637419765768007	0.5124241188709135,	0.08078480481085802,
	2	0.6781552335418927	0.25053489481136209,	0.7353191697604025,	0.5392100364203585,	0.08944732537687757,
	3	0.7503631013730631,	0.4311018731556629,	0.7094376648438742,	0.4968804491754838 5,	0.09331750160586859,
	4	0.7413115852390472,	0.3695921165508528,	0.7040600312037468,	0.5205667176199656,	0.10098209969142413,
	5	0.8009708009069129	0.3484533184668419	0.7107621007809871	0.4525731682782686	0.0885957568539273
	Av.	**0.704664188633162**	**0.3392562933380767**	**0.7170420414154092**	**0.5026483936847329**	**0.091648364873648689**

Code available in ModelEval.py

Figure 4: F1 score from five-fold cross validation performed on each model with Normalised Vectorizer (this is the vectorizer we prefer based on its additional functionality)

		Random Forest	Multinomial NB	Linear SVC	Decision Tree	Dummy
Normalised Count	1	0.3550516546981543	0.22208556076976363,	0.5888579451082016,	0.5023948755030261,	0.08111808316848422,
	2	0.3768175502474626	0.22588239125839635,	0.6743189928363641,,	0.5408989316476412,	0.0892537298302999,
	3	0.3628959674825088,	0.23200203840759953,	0.6295679600970382,	0.4878575642980717,	0.0939036639 5091897,
	4	0.3889902560014184,	0.23767845793838527,	0.5959596219498204,	0.5206046337079452,	0.1008269620 6685075,

	5	0.3581210994192518	0.23256125019245963	0.6153834781435680	0.4568026140307820	0.0884953879086488
	Av.	**0.368375305569759**	**0.230041939713320**	**0.620817599626998**	**0.501711723837493**	**0.090719565385040**

Code available in ModelEval.py

Figure 5: Recall score from five-fold cross validation performed on each model with Normalised Vectorizer (this is the vectorizer we prefer based on its additional functionality)

		Random Forest	Multinomial NB	Linear SVC	Decision Tree	Dummy
Normalised Count	1	0.3192144150948162	0.24182707508927226,	0.5405390344089931	0.49834772671706806,	0.08161738266219515
	2	0.3348124783288175	0.24686006027671106,	0.6329779159000694,	0.54646025549374 42,	0.08914486646920496,
	3	0.3233424055196794 7,	0.24879875545665975,	0.5847872186942163,	0.48286482914325 73	0.09469125828815894,
	4	0.3409252998373697,	0.25021876802200 01,	0.5433138091550 06,	0.52518293004469 95,	0.10092745920578562,
	5	0.3211157683662384	0.24745019714542274	0.5672358336595574	0.46597817374893713	0.08849594341402617
	Av.	**0.32788207342938425**	**0.24703097119 80132**	**0.57377076236 35685**	**0.50376678302 95413**	**0.090975382007 87417**

Code available in ModelEval.py

With the same feature vectorisation (Normalised Vectoriser), the Random Forest model achieved an ambiguous second position, at roughly 0.73 of mean accuracy. By inspecting other metrics that were calculated, we tend to observe a slight discrepancy in terms of the behaviour of the RandomForestClassifier() in light of its predecessor DecisionTreeClassifier(). However, RandomForest() classifier tends to reduce more of the variance part of error than bias.Though DecisionTree() is able to perform better in recall and f1, RandomForest() will almost always perform better in an unseen test set.

We decided to perform hyper-parameter tuning on these two models (RandomForest() and LinearSVC()), in hopes of improving these preliminary results even further.

Prior to commencing the training of our models, we had to optimise the configuration parameters of `TfidfVectorizer`. We found that `TfidfVectorizer` was better suited to our text classification model on the basis that it consistently provided better results than using the `CountVectorizer` in the table above (except Multinomial NB). Moreover, it has an inbuilt scaling feature that assists to balance the impact of various features when using L2 norm for penalty later in our LinearSVC() models. Furthermore, it provided us additional functionality in developing an optimal dataset by using the following parameters.

The most significant of these parameters include;

- `stop_words`, a list of basic and common words to ignore
- `ngram_range`, the maximum and minimum number of neighbouring words that can be used together as a single feature

- `min_df`, the lower threshold of frequency of articles containing each word, below which words will be excluded, and
- `norm`, the method by which the resulting frequencies are normalised

By using `stop_words='english'`, the number of unique words in the vocabulary is reduced from 35822 to 35725, a reduction of 97. Appendix 5.4.1 shows the full set of words that are excluded. Examples are 'but', 'how' and 'put', generic English language words that offer no classifying information. Ignoring them when vectorising helps to reduce the noise in the dataset, hence improving any model trained on it.

By using `min_df=5`, we were further able to get the number of unique words down to 9424. This pruning technique was used to remove words that have very few occurrences to be considered meaningful e.g. name of people in the article. Using `min_df=5` also helped us remove mis-spelled words, which could negatively impact the performance of the model. Thus, improve the overall performance.

The provided data has already experienced some amount of preprocessing, and due to the unordered and mangled nature of each word list feature, `ngram_range` is not applicable. `ngram_range` attempts to increase the number of classifying features by considering the frequency of combinations of neighbouring words, however we found that with unordered words it is not useful.

In the case of normalisation, the options are either least absolute deviations (`'l1'`), least squares (`'l2'`), or none, the default. This parameter adjusts the regularisation of each feature vector, and can have varying results depending on the dataset and models. We found that least squares produced the best results when combined with the most successful models in the benchmark test.

Our final vectorisation is `TfidfVectorizer(stop_words='english', min_df=5, norm='l2')`. This resulted in a significant decrease in the number of features we used and helped us not only perform better as determined by accuracy, but benchmark faster.

The first model we selected was the `RandomForestClassifier` from `sklearn.ensemble`. We decided to implement a cross-validated grid search for optimal hyper-parameters across wide ranges with low sampling, and repeatedly focus on a smaller range of increasingly optimal parameters. To save computation time, we identified the highest-priority hyperparameters and resolved to optimise the less significant configuration options individually once the most important ones were optimised together.

Our expectation is that at some point, increasing the number of estimators in the random forest will cease to increase the accuracy. This is because of the nature of the random forest classifier; it returns the majority classification across a wide population of individual decision trees, so any increase to the population has a diminishing return on accuracy at some number of trees. This was also demonstrated in the table below.

As increasing n_estimators had a diminishing return on accuracy, it should be safe to assume that increasing n_estimators from 100 whilst looking at the changes in accuracy when tweaking max_depths would be redundant. That being said, in the text file max_depth.txt it was shown that changing the max_depth won't make much of a difference on our accuracy, that being said theoretically reducing the max_depth below a certain point would me detrimental to the functionality

of the model as it may cause early stoppage, which is one of the limitations of pre-pruning decision trees.

Another factor parameter to take into account is min_samples_leaf. Results in the table show a gradual decrease in accuracy as we increment the minimum number of samples per leaf. Appendix 7.4.6 displays this descent graphically.

		min_samples_leaf						
		1	2	3	4	5	6	7
Accuracy	1	0.742105	0.742105	0.728947	0.731053	0.725263	0.723684	0.726316
	2	0.744211	0.734211	0.731053	0.725789	0.725263	0.721053	0.728947
	3	0.731053	0.733158	0.727895	0.728421	0.718421	0.721579	0.717895
	4	0.738421	0.735263	0.728421	0.727895	0.719474	0.720526	0.719474
	5	0.734211	0.732105	0.737895	0.729474	0.732632	0.721579	0.722105
	Av.	**0.738000**	**0.735368**	**0.730842**	**0.728526**	**0.724211**	**0.721684**	**0.722947**

In conclusion, we deemed `RandomForestClassifier(min_samples_leaf=1, n_estimators=100)` to be the optimal random forest for our purposes.

On the other hand, our second model was the `LinearSVC` from `sklearn.svm` where we performed a similar methodology as the `RandomForestClassifier`, implementing a controlled 5-fold grid search to determine the optimal parameters based on the measure of development set accuracy. We identified that the initial success of the SVC was possibly due to its ability to scale to high dimensions, as produced by our vectorisation. The methodology was kept constant between the `RandomForestClassifier and LinearSVC` as a means to promote fairness in aiming to compare the two models

Other variants of the support vector classifiers were also tested to observe if by increasing the complexity of the kernel from linear to polynomial, sigmoid or rbf, the accuracy of the model would improve. The optimal algorithm for classification is calculated through the `StratifiedKFold` object to be `linearSVC()`. The code is available in linearsvc.py. The 5-fold cross validation was primarily a means to improve the generalizability of the model lower and to be able acquire more metrics from our data like accuracy, recall, f1 and precision.

During our 5-fold `GridSearchCV`, we encountered the problem of certain combinations of the parameters not being viable to be tested simply due to the nature of the characteristics they bring to the model. For example, the combination of `penalty='l1'` and `loss='squared_hinge'` are not supported when `dual=True`, and as a result an error would be thrown. Hence, to overcome this issue we had to fix certain parameters as constants and create multiple instances of `GridSearchCV` for different variations of the same model.

Linear SVC Parameter tuning.

- `penalty` (L1 or L2) – Though L1 is better than L2 for high dimensional data for its sparsity, L2 was preferred to fix dual=True inorder to deal with a large amount of features. Moreover, setting `dual=True` required us to remain with L2 as our penalty for compatibility purposes within the grid search.

- `Loss` - After fixing L2 as `penalty`, performing a `GridSearchCV` revealed that the optimal parameter for loss is `'hinge'`.
- `Dual` – Though the recommended settings within the documentation of the `LinearSVC` class stated, "Prefer dual=False when n_samples > n_features", we still opted for the default `Dual = 'True'` for computation reasons despite n_samples > n_features. For our dataset, when `Dual='True'`, the computation time measured by `time.timit()` decreased by almost 100-fold (from 5073.95s when `Dual='False'` to 54.46s during a gridsearch). This allows the primal problem to be solved more easily when converted to its dual.
- `max_iter` - The best value for this parameter was narrowed down to 10 by performing a `GridSearchCV` for values of `max_iter` to be [10, 100, 1000, 5000, 10000] - from which we obtained 10 as our best value. We then commenced an accuracy test to determine 10 as optimal with parameters [1,5,10] being in our sample space. We discontinued to follow up on further narrowing the value due to high computation times.

4. Results

Cross-validation results on the training set for selected metrics, feature sets and implemented methods is available in appendix 6.4. Initial benchmark of SVC models showed clearly that LinearSVC was the superior model for this dataset. We decided to optimise the hyperparameters on that model to suggest the article recommendations.

Final results for each class calculated on the whole test set using `LinearSVC()` with its optimal hyper-parameters.

Results of metrics for different classes calculated by `LinearSVC()` model

Class	Precision	Recall	F1
Arts/culture/entertainment	0.33	0.67	0.44
Biographies/personalities/people	0.80	0.27	0.40
Defence	0.89	0.62	0.73
Domestic markets	1.00	0.50	0.67
Forex markets	0.46	0.33	0.39
Health	0.73	0.57	0.64
Irrelevant	0.95	0.97	0.96
Money markets	0.84	0.90	0.87
Science and technology	0.53	0.61	0.56
Share listings	0.00	0.00	0.00
Sports	0.50	0.43	0.46
Avg	0.639	0.53	0.556

Table generated based on code found in EvalMetric_classes.py

Recommended articles and per-class metrics

Category	Recommended articles	Precision	Recall	F1

ARTS CULTURE ENTERTAINMENT	9703, 9789, 9952	0.667	0.667	0.667
BIOGRAPHIES PERSONALITIES PEOPLE	9695	0.000	0.000	0.000
DEFENCE	9559, 9576, 9607, 9616, 9670, 9759, 9770, 9773, 9842, 9987	0.800	0.615	0.696
DOMESTIC MARKETS	None	0.000	0.000	0.000
FOREX MARKETS	9529, 9530, 9551, 9588, 9682, 9718, 9772, 9798, 9977, 9986	0.700	0.146	0.241
HEALTH	9609, 9661, 9807, 9833, 9873, 9926, 9929, 9937, 9947, 9978	0.600	0.429	0.500
MONEY MARKETS	9602, 9618, 9672, 9755, 9761, 9765, 9769, 9871, 9990, 9998	0.700	0.101	0.177
SCIENCE AND TECHNOLOGY	9617	0.000	0.000	0.00
SHARE LISTINGS	9518, 9601, 9666	0.667	0.286	0.400
SPORTS	9569, 9573, 9752, 9760, 9787, 9848, 9857, 9886, 9922, 9942	1.000	0.167	0.286

Obtained from `ArticleRecommendation.py`

5. Discussion

Accuracy is a prominent classification metric due to its ability to provide an easy interpretation and straightforward comprehension of our model. Nevertheless, this will not be so useful as a predictive measure in a dataset that is obscurely skewed towards a specific subset of the classes - in our case ‘irrelevant’. The percentage distribution being as high as 49.8% of the number of instances (4734 of the 9500 instances), normal algorithms tend to train on this dataset to output a bias towards the ‘irrelevant’ label. Our least favourite scenario now would be to represent the rare classes to be anomalies and disregard them. Therefore, to account for this effect we implemented a cross-validation algorithm using stratifiedKfold which provides us a good generalization of the data and at the same time lower our variance. Using this algorithm, we calculated average precision, f1 and recall.
The table in appendix 7.3.1 depicts a summary of these metrics. We can clearly see the dominating model of the two being linearSVC. These measures communicate that the features brought by LinearSVC() are more relevant in handling both dense and sparse inputs, perhaps due to its flexibility in the choice of penalties and loss function - allowing it to scale better to large numbers of samples.

In examining the results from precision (`Precision = (TP)/(TP+FP)`), we can clearly identify that the proportion of predicted classes that are truly part of the predicted class is only slightly higher in the LinearSVC() (0.71) in comparison to its opponent RandomForestClassifier() with a score of 0.70. This reveals that precision is not a good measure in determining our superior model out of the two best models. Though it does provide us the means to separate these two models from our initial bag of five classifiers in light of the small difference between our 2 main models (the other models having scores of 0.339, 0.50 and 0.090).

Subsequently, we obtained recall scores (0.386 for RFClassifier() and 0.574 for LinearSVC()) in-order to determine how many instances were being correctly captured. Upon inspection we concluded again that LinearSVC() has the upper hand in returning most of the relevant results in comparison to its counterpart.

Furthemore, to examine how our models perform in the light of both precision and recall simultaneously. We wanted to compare our tuned model to our untuned model to observe an increase in this quantity - validating an improvement in both recall and precision.

Metrics for the above table.
The table with final article recommendations and accompanying metrics exhibits both some advantages and disadvantages of the evaluation strategies employed.
Precision offers a quick and relatable insight into the quality of recommendations given to a user. A category with a precision closer to 1 represents a very accurate selection of articles, and as such the most successful categories can be easily identified by their high precision scores.

On the other hand, precision does not account for categories which had no true positive recommendations. This would be problematic, for example, if a hypothetical category was present at large in the training set but had no articles in the test set. In this example, recommending no articles would be the best recommendation, but would not be reflected in the precision score, which would be 0.

This issue is also present in the other metrics used to evaluate the results. A recall of 0 holds little value, as it is not indicative of how many true positives were missed, just that none were found. Because F1 is a derivative of precision and recall, it also offers no further indication of performance when the recommendation is empty.

A different, but very damning disadvantage is found in the calculation of recall. The number of recommendations is limited to 10, so the number of true positives could only be 10. However, the number of all positives is comparatively unbounded, leading to low recall on, for example, the SPORTS recommendations, despite a high recall when considering the entire category.
This could be remedied by modifying the recall calculation. The denominator should be either the number of all positives in the dataset, or 10, whichever is smaller. This means that recall will always be given by the ratio of how many true positives were selected to the total selectable true positives.

With this modified recall metric, a model won't be punished for correctly selecting 10 articles, regardless of how many correct articles are available. Other cases, such as selecting more than the number of positives in the dataset, still yield results that indicate performance under this new metric.

In continuing this project, some modification in the methodologies section would be to use RandomizedGridsearchCV() before looking for optimal parameters by using GridSearchCV() to focus on a closer range of values that contribute to optimizations. Likewise, we could implement a confusion matrix and zoom in on which article was getting mis-classified the most and try to find ways to improve the models performance in that specific area. Also, we could analyse whether it is high precision or high recall that is contributing to the increasing efficiency of our algorithm and thereby create a weighted F1 metric to manage the tradeoff between the two metrics.

Moreover, in terms of analyzing the RandomForestClassifier() algorithm, using the `oob_score` parameter to set it at True and False in order to experiment with this parameter to obtain better results. On the other hand, for LinearSVC(), minimizing scores of log loss cross entropy inorder to increase accuracy for our best classifier would be more apt for usage in our multi-label data.

GridsearchCV was mainly a problem due to a compatibility issue that occurred during the development of the model for LinearSVC(). We overcame it by creating multiple instances of the same algorithm with different parameters to observe for optimisation gains.

6. Conclusion

The problem we sought to solve was multi-faceted. We needed to find a reliable way to extract features from the text provided, and then compare the efficacy and applicability of different evaluation metrics, apply them during cross validation to optimise multiple models, and finally use the most effective model to produce confident classifications of unseen test data.

We learnt the value of investigating the defaults provided by sklearn, and evaluating ourselves what the needs of our project were. The metrics used in scoring by default often did not provide enough information, and we learnt through the project that doing manual analysis was just as important.

In a similar tune, although the library is powerful, we learnt that time constraints are very, very prominent. Using intelligent hyperparameter options for a grid search is important, because without smart selection, you are faced with a search that runs for hours and often even results that are hard to interpret.

We also learnt specific details of each hyperparameter available to us. We all understand the options provided by sklearn in vectorisation, normalisation, and training amongst many models. This knowledge helped us choose hyperparameters to focus during the project.

By taking initiative with our parameter selection and allocating computation tasks between group members carefully and cleverly, we achieved a large amount of computation across a vast range of search parameters in a short amount of time.

The science behind data science became very clear. After the days and weeks of work reasoning between ourselves, running large searches, and scanning sklearn documentation, we saw the fruits of our labour when we observed the high accuracy achieved by our final model. We achieved a surprisingly good result.

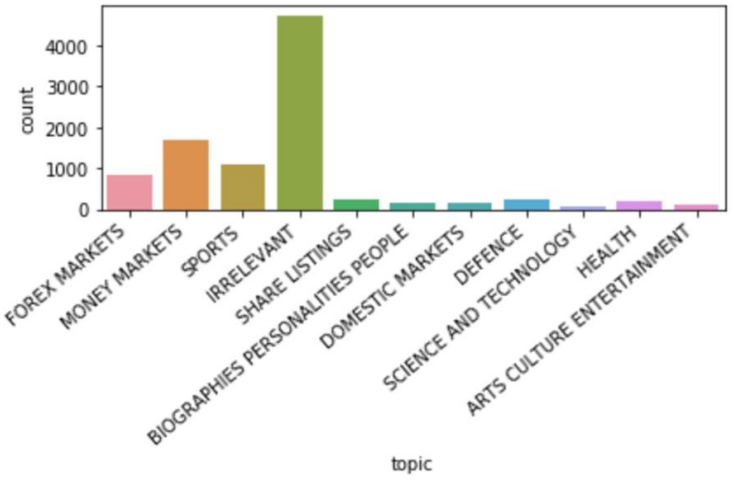
7. Appendix

7.1 No appendix items for introduction

7.2 Exploratory data analysis

7.2.1

Number of instances of each class within the dataset.



7.3 Methods

7.3.1

Cross validation results of metric evaluation for different models after hyperparameter optimisation

	`RandomForestClassifier`	`LinearSVC`
Mean Accuracy	0.732	0.770
Precision	0.705	0.717
Recall	0.328	0.574
F1	0.445	0.621

7.4 Results

7.4.1

Words excluded from the dataset by using `stop_words='english'`

```
Excluded words:{'fifty', 'how', 'against', 'mine', 'three', 'less', 'but', 'much',
'such', 'found', 'two', 'put', 'further', 'amount', 'interest', 'show', 'being',
'bill', 'do', 'may', 'must', 'sixty', 'ever', 'first', 'bottom', 'should', 'made',
'see', 'both', 'top', 'forty', 'four', 'eight', 'call', 'thin', 'them', 'move',
'empty', 'your', 'own', 'off', 'and', 'thick', 'per', 'inc', 'give', 'detail', 'side',
'cant', 'third', 'down', 'fill', 'might', 'even', under', 'fire', 'over', 'ten', 'any',
'except', 'will', 'with', 'part', 'out', 'nor', 'whole', 'six', 'last', 'some', 'take',
'name', 'find', 'not', 'get', 'mill', 'five', 'system', 'fifteen', 'all', 'back',
'can', 'full', 'well', 'her', 'due', 'hundred', 'keep', 'nine', 'more', 'had', 'de',
'than', 'cry', 'front', 'few', 'con', 'seem'}
```

7.4.3

5-fold cross validation mean accuracy for SVCs

		`SVC(kernel=)`		
	`LinearSVC`	`'poly'`	`'sigmoid'`	`'rbf'`
1	0.7710526315	0.6657894736	0.75631578947	0.76947368421

	789473	842105	368421	05263
2	0.771052631578947	0.657894736842105	0.751578947368421	0.772631578947368
3	0.771578947368421	0.66	0.768947368421052	0.756315789473684
4	0.764736842105263	0.671578947368421	0.760526315789473	0.761578947368421
5	0.772631578947368,	0.661578947368421	0.766842105263157	0.76
Av.	**0.7702105263**	**0.6633684211**	**0.76084210526**	**0.764**

Table generated through code found in `svc_model_comparison.py`

7.4.4

Mean accuracy of 5-fold cross validation, with `loss='squared_hinge'`

		C										
		1e5	1e4	1e3	1e2	1e1	1e0	1e-1	1e-2	1e-3	1e-4	1e-5
max iter	1	0.75	0.776	0.788	0.778	0.716	0.766	0.748	0.728	0.522	0.522	0.522
	5	0.782	0.774	0.77	0.774	0.768	0.778	0.768	0.706	0.522	0.522	0.522
	10	0.766	0.764	0.76	0.754	0.772	0.778	0.768	0.706	0.522	0.522	0.522

Table generated from code found in linearsvc.py

7.4.5

Mean accuracy of 5-fold cross validation, with `loss='hinge'`

		C										
		1e5	1e4	1e3	1e2	1e1	1e0	1e-1	1e-2	1e-3	1e-4	1e-5
max iter	1	0.784	0.792	0.784	0.774	0.762	0.794	0.746	0.556	0.522	0.522	0.522
	5	0.758	0.768	0.78	0.778	0.762	0.794	0.74	0.536	0.522	0.522	0.522
	10	0.76	0.76	0.75	0.772	0.772	0.796	0.742	0.536	0.522	0.522	0.522

Table generated from code found in linearsvc.py

7.4.6