1 Article Category Classification

In this part, we present the approach we followed to deal with the first part of the assignment. Our target was to create models able to correctly classify articles belonging in one of the following categories, *entertainment, business, health, technology*. In order to successfully achieve our target we have to follow a series of steps which help us avoid common machine learning pitfalls.

First, we got a feel of the data, helping us understand the problem we are trying to solve. At the same time, we briefly look for any "quirks" in our corpus like the usage of smart quotes (“, ”), which might not get caught when the punctuation is removed in future steps. We then established a baseline score by minimally cleaning the text using the suggested steps given in the assignment, representing our data in a Bag of Words manner, and fitting default SVM and Random Forest classifiers.

Having established a benchmark score we experiment with methods of improving our accuracy like hyper-parameter tuning, utilisation of the title but most importantly a more complex preprocessing method which aims at retaining as much information as possible.

1.1 Feeling the Data

We first have to get to know what data we are given in this classification problem. As a first step, we open our train dataset and randomly skim through some texts.

Example Articles			
Id	Title	Content	Label
48507	TweetDeck Briefly Shuts Down in Response...	the service was shut down for an hour as tweetdeck...	Technology
234755	Jon Hamm talks 'Mad Men' role...	despite "mad men" creator matt weiner's penchant...	Entertainment
93740	MERS death toll surges to 282 in Saudi...	baku-apa.??saudi arabia's death toll from...	Health
175422	US STOCKS-Wall St slumps in broad decline...	*dow back under 17,000 in broad market decline*...	Business

Table 1: Random articles from each one of the four classes.

In *Table 1* above, we can see how our texts are stored in the given train data file. We can see that while the title seems to not have any preprocessing applied, the content of the text is lower-cased. This is slightly unfortunate since we could have gained some information by proper nouns like names. For example, we would expect the entertainment articles to include more names in their content when compared to

the health articles.

We then check the number of instances of each category.

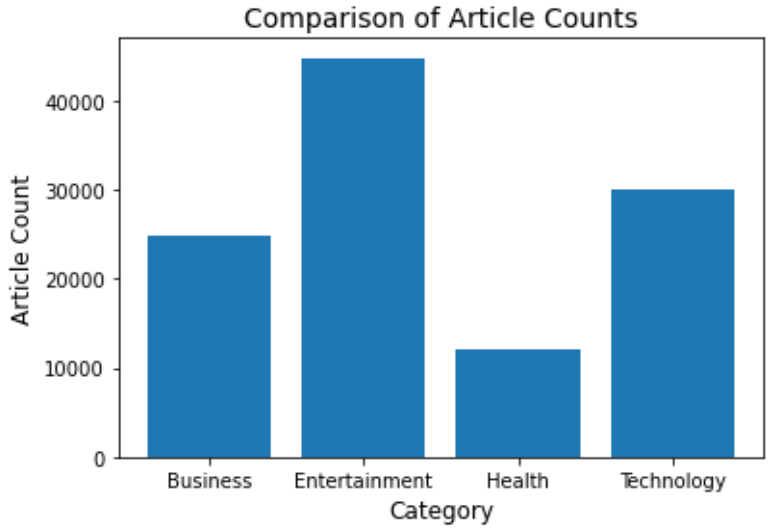

Figure 1: Number of data points of each class in our labelled dataset.

In *Figure 1*, we can see that the dominant class is *entertainment* while the *health* one has the smallest number of articles, being around 4 times smaller than the dominant class. As we progress, we should take this under account since our dataset seems to be slightly imbalanced. However, 10,000+ articles intuitively feel like enough to describe the *health* category, which we believe has a clearer meaning when compared to *business* and *technology*.

We then generated a word-cloud for each one of the four classes. In order to make it easier to create insights from the word-clouds we filtered any word included in a set of English stopwords and also manually removed words that seemed to appear with high frequency **in all of the categories**. These words were *"said, one, also, may"*.

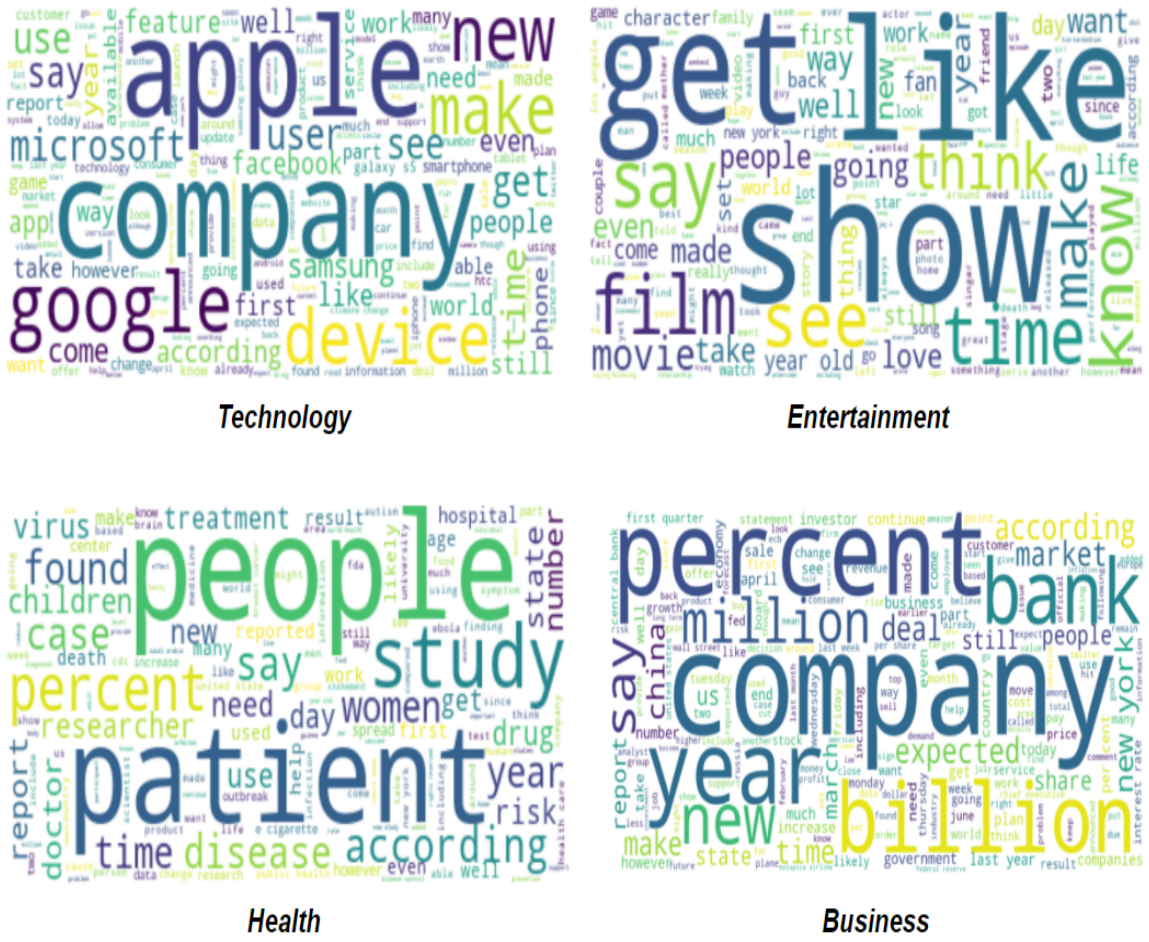

Figure 2: Word-clouds of each category.

Observing the word-clouds we understand that the health and entertainment categories will be easy to classify correctly since their words with high frequency (big font-size in the word-clouds) are not common in the other classes. However, this is not the case with the business and technology classes which share a lot of dominant words such as "company" and "make". Admittedly, even we as humans would probably have a hard time correctly classifying some articles in one of the two "difficult" classes. For example, the sentence "Facebook acquires OculusVR for $2 billion" could be classified as both business and technology.

1.2 Data Preprocessing

Cleaning the texts is a necessary step in order to create a refined representation of the contents and the titles that will then be vectorized. The cleaning depends on the data and the task given. We propose two different methods of cleaning the text. The first one is simple and will be used to create the benchmark scores. The second one is more complex and is used in our attempts to beat the benchmark and get the highest position in the Kaggle competition.

The steps of the *baseline text cleaning* are:

1. Lower all the letters
2. Remove stopwrods
3. Remove punctuation
4. Number removal
5. Lemmatization
6. Stemming
7. Frequent / rare word removal

The order matters since the stopwords include words like "doesn't" so if we had removed the punctuation first, the word "doesn't" would not be removed. However, we immediately notice a lot of information getting wasted. For example, all the numbers are removed but if a text includes many dates or money amounts, this may be a hint on which category it belongs. Also, a text containing open and closed quotes (“,”) would be less probable to be a health article. We present all the information that we utilize in the following list of steps. Again, the order is important.

The steps of the *refined text cleaning* are:

1. Lower the words
2. Find and replace urls with *"HTTPSYMBOL"*
3. Find and replace dates with *"DATESYMBOL"*
4. Replace money amounts (eg. US$12.3, 122,222$) with *"MONEYSYMBOL"*
5. Replace the utf apostrophe character (’) with an ascii apostrophe (')
6. Replace smart open quotes (“), close quotes (”) and the usual quote (") with *"QUOTESYMBOL"*
7. Replace left and right parentheses with *PARENTHESISSYMBOL*
8. Remove stopwords
9. Replace remaining numbers with *"NUMBERSYMBOL"*
10. Replace the ellipsis punctuation (..., . . .) with *"ELLIPSISSYMBOL"*
11. Remove any remaining punctuation

12. Lemmatization of each word
13. (Optional) Stemming of each word

Also, we have extended the stopwords set to include the words from the `stop-words` package along with the `nltk` words.

We apply the same cleaning methods on both the title and the content of all the articles in the train and the unlabelled test set and save them in the disk in order to avoid repeating these steps again.

1.3 Bag of Words Representation

In this part we have to transform our preprocessed texts from a series of words to a number vector which is the expected input for our machine learning algorithms. One main representation is the Bag of Words approach where we care about the frequencies of each word in each text and not the order that they appear.

We briefly present three bag of words techniques:

- **Count Vectorizer**, counts the number of each word appearing on a specific text. The main disadvantage is that it ends up giving higher weights on longer texts since they include more words.
- **Term Frequency (TF)**, deals with the above disadvantage by normalizing the appearances of words in a text by its length.
- **Term Frequency - Inverse Document Frequency (TF-IDF)**, in addition with the normalization above it also gives a higher weight to rarer words and lower weight to more frequent words. This allows us to not have to remove any frequent/rare words.

We decided to use the *TF-IDF* approach since, when we attempt to beat the benchmark, we might want to remove any frequent or rare words in the preprocessing steps (the `max_df, min_df` parameters). This might increase the computation time, but it retains all the information. If a word is too frequent our classifiers will ignore it without having us to remove it "manually".

By applying the vectorizer with a max-frequency of 0.8 and a min-frequency of 0.01 on the whole training dataset we create a sparse matrix of dimensions $111,795\ x\ 2,961$ meaning 111,795 texts and 2,961 features (vocabulary) in the case of the baseline cleaning method. In the case of the refined cleaning method we create a matrix of dimensions $111,795\ x\ 453,617$. The huge increase in features comes mainly from the fact that we do not use any limits in term-frequency and also the title utilisation method we presented above.

Also, we must note that one can reduce the dimensions of the dataset using **Singular Value Decomposition (SVD)**. While usually this means a loss of information, in our case the matrices are sparse, allowing us to reduce their feature dimension with a relatively small loss in variance. Specifically, we reduced the dimensions of the benchmark dataset from 2,961 to 1,000 while keeping 71% of the variance. However, in the final part, where we try to beat the benchmark, we did not reduce the dimensions since the `LinearSVC` implementation we used is optimized for sparse matrices.

1.4 Establishing a Benchmark

In this part we establish a benchmark score using the default SMV and Random Forest classifiers. We feed into these models the baseline dataset which was created with the simple cleaning method. Also, we compare the results using the 2,961 vocabulary dimensions and the truncated 1,000 dimensions.

Support Vector Machine - SVM

A support vector machine (SVM) is a supervised machine learning model aiming to find a separating plane with the biggest margin between two classes. Of course, in our use case we require multi-classification so the OneVsRest or the OneVsOne strategies are employed.

Random Forest

Random forest classifier is an ensemble of decision tree classifiers. While a decision tree is powerful at fitting on our training data, it is also prone to overfitting. By ensembling a set of decision trees, each of which can be regularized (e.g. limit their depth), we can battle the overfitting of a single deep decision tree.

In order to correctly evaluate the results of the above models, we apply k-fold cross-validation with 5 folds and report back the average of a set of metrics like accuracy, precision, recall and f1-score. In the case of recall, precision and f1-score we use the macro-average which first calculates each metric for each class separately and then averages them.

Statistic Measure	SVM (BoW)	Random Forest (BoW)	SVM (SVD)	Random Forest (SVD)
Accuracy	0.94	0.93	0.94	0.93
Precision	0.94	0.93	0.93	0.93
Recall	0.93	0.91	0.93	0.92
F1-score	0.94	0.92	0.93	0.92

Table 2: Benchmark metrics using 5-fold cross validation.

In *Table 2*, we can see that we achieve an accuracy of around 0.94 on all the methods, with and without SVD dimensionality reduction. Now, having established

a benchmark score, we are going to attempt to beat it using a set of methods and techniques that we have already presented (refined text cleaning, title utilisation) and we summarize in the following section.

1.5 Bayesian Optimization and Ensembling

For each classifier we tune, our training process is as follows:

To tune our models' hyperparameters, we employ Bayesian Optimization. It is a technique which probes points in the defined hyperparameter space and uses knowledge of the previous probes to fit a posterior distribution on our performance. This allows us to reach a faster and better maximum than what Grid Search or Random Search would achieve.

During training, we record all models produced - regardless of method -, their hyperparameters, and their validation predictions. Afterwards, we use all available models from all methods to form an ensemble which maximizes validation performance. To achieve this, we use the algorithm suggested by Feurer et al. [1] to combine them into an ensemble greater than its parts.

Starting from just the highest-performing model, we iteratively, and with repetition, add to the ensemble that model which increases validation accuracy the most. We continue this process until either our maximum number of ensemble members is reached, or until no model can further increase our validation accuracy when added to the ensemble. For the rest of this report, we will refer to the ensemble produced this way as our final model.

1.6 Beating the Benchmark

In this section we present and summarize our attempts at beating the benchmark score established above.

Refined Preprocessing

We changed the way of the preprocessing, aiming to keeps as much information as possible by parsing dates, money amounts, useful punctuation etc. We presented the new preprocessing technique in section 1.2. Of course, this increased the number of features and along with not removing rare words, ended up in a huge 453,617 dimensions feature space.

LinearSVC Implementation

Briefly looking on the documentation, we would expect the SVC with linear kernel and the LinearSVC of scikit-learn to be the same. However, we noticed that while the

fit time for SVC with linear kernel was around 40 minutes on 0.8 of the train dataset the LinearSVC needed only 1 minute making it an order of magnitude faster on the exact same data. After looking into their implementations we found out that LinearSVC is a liblinear estimator meaning that it penalizes (regularizes) the intercept while SVC is a libsvm estimator without any regularization on the intercept. Also, LinearSVC in the case of multiclass classification uses the OneVsRest technique while SVC the OneVsOne. Finally, the liblinear is able to scale much more efficiently when compared to libsvm especially on sparse matrices. This efficiency on sparse matrices constraints us to not use any dimensionality reduction techniques (like SVD), since the reduced matrix would be dense.

Title utilisation

The titles certainly offer us some information that could help our models achieve better accuracy. One straight-forward approach could be to concatenate the title and text. However, using only this approach does not exploit the title's significance. For example, if we saw the word "deal" in a title we would assume with great probability that this is a business article. However, the word deal in the content of an article could mean with higher probability the action of confronting. The example may not be perfectly accurate, but we believe it conveys the motivation of the technique we followed in addition to the simple approach of concatenating it at the end of the texts.

We chose to add a prefix on every word of our cleaned dataset, depending on whether the word is in the title or in the content. For example the article with **title** `"dog eats food"` and **content** `"dog eats food then the cat firmly complaints"` would become `"TITLE_dog TITLE_eats TITLE_food TEXT_dog TEXT_eats TEXT_food TEXT_then TEXT_the TEXT_cat TEXT_firmly TEXT_complaints"`.

N-grams

In our case we define n-grams as the sequential combination of words. The length of each combination is n and is given as a parameter. For example in the case of $n = 2$ (bigrams) , the phrase `"The dog eats food"` would create the sequences `"The dog, dog eats, eats food"`. We experimented with the n parameter using cross-validation, and found that combining the unigrams(n=1) and bigrams gives us the best validation accuracy.

Table 3, suggests that the best classifier is a combination of the refined preprocessing, title utilisation and n-grams (unigrams combined with bigrams). Now, we tune our best - so far - classifier in order to get the most out of it.

Finetuning

In order to get the most out of our classifier we tuned its hyperparameters using

Statistic Measure	Refined Preprocessing	Title Usage*	N-grams[1-2]*	N-grams[1-3]*
Accuracy	0.9708	0.9748	0.9726	0.9701
Precision	0.9694	0.9734	0.9718	0.9695
Recall	0.9671	0.9719	0.9684	0.9652
F1-score	0.9682	0.9726	0.9701	0.9673

*Includes the refined preprocessing too.

Table 3: Metric improvement on the validation set (5-fold) without any hyper-parameter tuning.

Bayesian Optimization.

In the case of LinearSVC, we tuned its regularization parameter C, its tolerance, its loss function, the choice of solving the primal or the dual optimization problem and the choice of balancing or not the weights of its class since we have a slightly imbalanced dataset. Afterwards, we used the ensembling algorithm described above to combine the produced models into an ensemble.

The final model (*Table 4)* with tuned hyperparameters, achieved an accuracy of 0.9772 using 5-fold cross-validation and the same accuracy on the Kaggle competition.

Statistical Measure	LinearSVC
Accuracy	0.9772
Precision	0.9762
Recall	0.9750
F1-score	0.9740

Table 4: The best metrics achieved after hyper-parameter tuning.

1.7 Possible Optimizations

There is definitely room for improvement and we present possible ideas and follow-ups we had that could lead to accuracy improvements. Firstly, we created a confusion matrix (*Figure 3*) aiming to show us the weaknesses of our currently best classifier.

We can see as expected that the most misclassified classes are *Business* and *Technology* ,which intuitively makes sense since they have an intertwined meaning. Some possible solutions we thought of were:

- Boosting, which will focus on the articles that are mixed between *Business* and *Technology*
- Usage of neural networks like RNNs or even tuning a pretrained transformer model such as BERT/GPT. Admittedly, this seems like it would not fit our

use case, since the content of an article mostly defines its category and not its syntax / grammar that these networks are efficient at understanding. Also, the size of the articles prohibits the usage of these techniques.

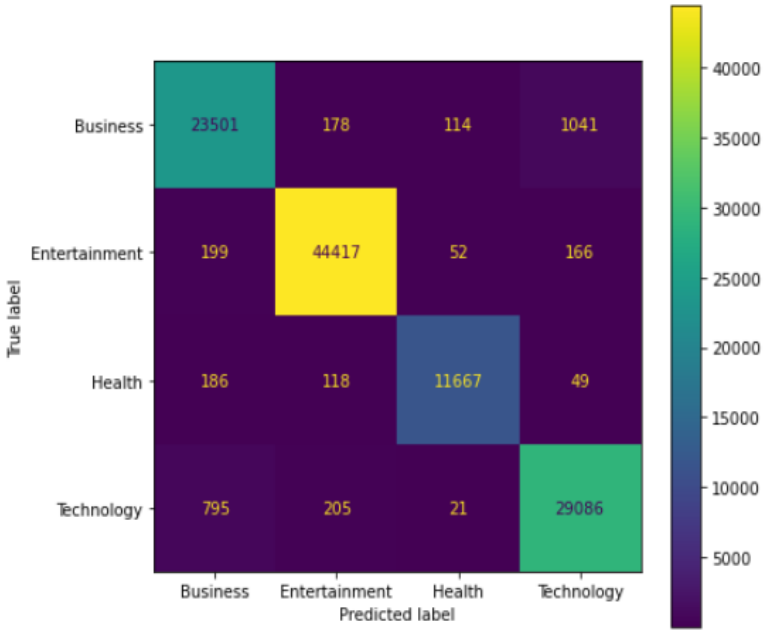

Figure 3: Confusion matrix of the currently best classifier.