Kaggle Data Competition 1 : Classification task

Daniel Galarreta-Piquette, Team Name : Daniel GP

1 Introduction

For the *IFT6390* class, we did a Kaggle data competition where we had to solve a text classification task. The goal was to predict the class of submitted papers from their abstract. It was required to solve this problem using a Naive Bayes Bernoulli classifier, with sole access to standard Python libraries (except for pandas and numpy). We had to use a bag-of-word (BOW) model in order to vectorize text and do some feature selection and hyper-parameter optimization. After having polished our Bernoulli NB classifier, we could use any other algorithm from another family of functions and solve the task without constraint on Python libraries.

Our second experiment was with Support Vector Machine (SVM), but noticing that the new algorithm only beat the previous one by a margin, we ended up going back to Naive Bayes but with Multinomial approach. In the end our best model was the NB Multinomial with an 80% accuracy, while the SVM did 79.1% and the Bernoulli NB was a third place with 77.8 % .

2 Dataset and metrics

For this competition, we had access to the following data : 7500 abstracts from arXiv. The data was evenly distributed, exactly 500 papers per class. Some classes look similar (4 variants of astro, 2 variants of math, etc.) in name, which means that they might share the same kind of vocabulary. Since we use the vocabulary to predict a class, we might mixup predictions between similar classes the wrong label between two classes that are similar. We will come back to this in section 6 and 7.

For this task, the performance of the classification models was evaluated using accuracy (i.e. : the number of correctly classified instances divided by the total number of instances).

3 Feature Design

3.1 Bernoulli Naive Bayes

For the Naive Bayes Bernoulli we used the following pre-processing.

- We stripped trailing white spaces
- We replaced breaking lines with spaces
- We used regular expressions to clean the data from special characters
- We removed non-ascii characters and keep the data in utf-8 encoding
- We tokenized each abstract
- We lower-cased every token
- We removed stopwords tokens
- We made sure that we only kept alphabetical strings

Cleaned and normalized data will more likely improve performance because we are getting rid of *noise*. We will be training our models with a bag-of-word method which uses the vocabulary as features (or dimensions). If we only keep meaningful features, we will drastically reduce process time and more likely improve accuracy. For example, by lowercasing all of our tokens we got rid of doubles : *Model* and *model* would be normalized to *model* for example.

With a BOW method for transforming data into vectors, all of the words from all the abstracts are used to build the vocabulary. It comprises of single occurrence of all the unique words in our dataset. We will vectorize each data point (abstract) by assigning a count in a column if a certain word appears in a given abstract. We do this for all of the words in each abstract. So in the end, we will end up with a vector that has the lenght of the vocabulary. It will be a sparse matrix composed of 0 and digits. (0 if a word is not in the abstract and the right count for the number of times that word appeared in the given abstract. This is the corner stone of our algorithm. We use these vectors to correctly (or wrongly) classify unseen data (our test data).

Our processing and accuracy will depend on the dimensions that we decide to keep. Without any cleaning, we had a vocabulary of more than 20 000 tokens post pre-processing. It is true that more dimensions might help the model to better generalize because it has more information, but sometimes too many dimensions worsen the accuracy because the learning algorithm will not be able to generalize (the famous curse of dimensionality). So we need to find the *sweet spot* : the right amount of dimensions that will make our loss function at its lowest point.

So we need to reduce our vocabulary to meaningful features. We could filter the tokens based on a certain constraint like a minimum count. We saw that many words appeared only once, we might want to reject those words. If they appeared only once, the odds that they will help us generalize on unseen data is very low. They are most likely slowing the process and not helping us recognize similar texts between the same classes.

Also, in the spirit of reducing the number of dimensions, we decided to implement a way to normalize plural or 3^{rd} person present forms to their singular counter-part (or infinitive). Knowing that lemmatization is often used in Natural Language Processing, we thought that this basic implementation might maximize our accuracy. So, if a word exists and

its equivalent with an "s" at the end also exist, remove the one with the "s" from the vocabulary.

3.2 Other models : Multinomial NB and SVM

For this part, we used Spacy's pipeline for text classification[1]. The two other models that we tested were Multinomial NB and Support Vector Machine. Both of these had the same pre-process and feature engineering. Using Spacy's english parser, we parsed every abstract, and then we tokenized all the strings and we made some adjustments. We lowered all cases, removed punctuation and stopwords, lemmatized, and removed symbols and numerical tokens. We then transformed each datapoint into a vector by using two variants, the BOW and the TF-IDF. Those will become parameters for the Multinomial and SVM approach. In order to clarify this parameter, we must briefly explain what is TF-IDF (term frequency–inverse document frequency). It's an alternative method to BOW, because the output is also a sparse vector. But instead of counts, we end up with weights. We will assign heavier weights for words (features) depending on their frequency and usage in the corpus.

Finally, the two last parameters that had an impact on feature selection were the length of n-grams and the minimum frequency in corpus. For the length of n-grams, the idea is to create more dimensions that will be more contextual, because we will consider the word before and the word after for each of the tokens in the vocabulary. Thus, this will create more dimensions, but maybe meaningful ones. We decided to test up to a range between unigrams and trigrams. Now, for the minimum frequency in corpus, it's basically the same parameter that we stated earlier in our Bernoulli NB feature selection. We can put a minimum constraint for a word to be considered in our model.

4 Algorithms

In this section we will present the three algorithms we decided to use. Two of them are from the same branch so we will explain them in the same section, namely the Bernoulli and Multinomial issued from the Naive Bayes family of algorithms. Afterwards, we will describe the third algorithm we tested which is the Support Vector Machine (SVM).

4.1 Naive Bayes

The Naive Bayes classifier is a probabilistic algorithm that emanes from Bayes Theorem. Basically, it models the conditional probability of observing the dimensions (or features) of a given test data X, assuming that it belongs to Class K. The class that will be given as label to the test point X, will be the one with the highest probability, given our prior.

$$\mathbb{P}(C_k|X) = \frac{\mathbb{P}(X|C_k)\,\mathbb{P}(C_k)}{\mathbb{P}(X)}$$

$$\hat{C} = \arg\max_{C_k} \mathbb{P}(X|C_k)\,\mathbb{P}(C_k)$$

It is important to note that we also assume that the features of X are independent inside of class K.

4.1.1 *Multinomial Naive Bayes*

This version of the NB classifier makes the assumption " that the probability of each word event (occurrence) in a document is independent of word's context and position in the document. Thus, each document can be represented as a vector of word counts, x, and its class-conditional probability is given by a multinomial distribution over the set of words. "[1]

$$\mathbb{P}(X|C_k) = \mathbb{P}(x_1|C_k) \times \mathbb{P}(x_2|C_k) \times \ldots \times \mathbb{P}(x_d|C_k)$$

If a conditional probability $\mathbb{P}(x_i|C_k) = 0$, that is because x was not included in the training dataset for the given class. The consequence of multiplying by zero makes it so the class-conditional probability collapses to 0 without taking into account the rest of the probabilities in the chain rule. To correct this problem, we regularize the estimated point $N_{j \text{ in class } C_k}/N_{\text{total}}$ with α , we call this smoothing, and the new point estimated corresponds to $\frac{N_{j \text{ in class } C_k}+\alpha}{N_{\text{total}}+\alpha}$.

4.1.2 *Bernoulli Naive Bayes*

The Bernoulli distribution corresponds to

$$\mathbb{P}(X|C_k) = \prod_{i=1}^{n} p_{ki}^{x_i}\,(1-p_{ki})^{1-x_i}$$

When this algorithm is used in text classification,it uses the binary occurrence information, which means that we are only interested in 0 counts in the vector or an occurence (equivalent of 1). Thus ignoring the number of occurrences, instead the multinomial model keeps track of multiple occurrences[2] We used a binarize implementation in our Bernoulli NB that would function like a switch where we attribute 0 if the feature is not present in the text or 1 as soon as there is a count. This way, our method of generating bag-of-word could be used with other algorithms that are interested in counts.

[1] https://www.dataquest.io/blog/tutorial-text-classification-in-python-using-spacy/.

[2] https://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html.

The implementation that was used came from a tutorial that explained how to create Naive Bayes classifiers[3].

4.2 SVM Classifier

We used a one-vs-rest (OVR) K-class classifier. We created K binary classifiers that will divide the training data-points belonging to class k on one side of the decision boundary in the feature space, and the rest of the training points that are from all the other classes will be put on the other side of the hyperplane. The hyper plane is an affine function in the feature space that divides the data into two classes (one and the rest). We call it Support Vector Machine because the points that interest us are the ones really close to the decision boundary, because they are meaningful to help us classify unseen data. Points that are really far from the decision boundary are not that useful because they do not allow us to see which features distinguish one class from the rest.

This classifier aims to learn K hyperplanes defined by an affine function that maximizes the margin from the decision boundary between the kth class and all the other possible classes.

The SVM has a regularization term called C that controls the tolerance to test points that were mis-classified. The bigger the value of C is, the bigger the tolerance to mis-labelling. There partly lies our bias-variance tradeoff.

Using Scikit-learn's SVM implementation, we could try different configuration of the model. If it seems that the data points are not linearly separable, a *kernel trick* can be implemented to project the data points in a higher dimension space; the decision boundary is not linear anymore $.K\left(\mathbf{x}_i, \mathbf{x}_j\right) = \langle\phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ where $\phi(\mathbf{x})$

"We replace all dot products with a kernel function K. This corresponds to running the algorithm in the feature space without ever having to explicitly compute the mappings with the feature map ϕ"[2]. What happens is this :

$$
\begin{aligned}
g(\tilde{\mathbf{x}}) &= \langle\tilde{w}, \tilde{\mathbf{x}}\rangle + b \\
&= \langle\tilde{w}, \phi\left(\mathbf{x}\right)\rangle + b \\
&= \langle\sum_{i=1}^{n} \alpha_i \phi\left(\mathbf{x}_i\right), \phi\left(\mathbf{x}\right)\rangle + b \\
&= b + \sum_{i=1}^{n} \alpha_i \langle\phi\left(\mathbf{x}_i\right), \phi\left(\mathbf{x}_j\right) \\
&= b + \sum_{i=1}^{n} \alpha_i K\left(\mathbf{x}_i, \mathbf{x}_j\right)
\end{aligned}
$$

There are different types of kernels that we tested with SVM, the three implementations were: linear, polynomial and gaussian (RBF). We will not go into details here, but the linear kernel is good when the data is linearly separable. It is also the most interpretable one and easier to implement. The RBF and polynomial kernels will elevate the number of dimensions and this will permit the decision boundary to be non-linear and adjust to the data that was not linearly separable initially.[2].

[3]https://kenzotakahashi.github.io/naive-bayes-from-scratch-in-python.html.

5 Methodology

All models were trained on a 70-30 split, and all parameters and hyper-parameters were chosen based on a 5-fold cross-validations for each different change. All of the accuracies are averages of 5-fold cross-validations.

5.1 Bernoulli NB

After having completed our feature selection and pre-processing phase, it was time to validate which parameters and features will end up minimizing the loss on our validation set. We started by finding what minimum frequency of a token was needed in order to boost our performances. By finding that threshold, we reject all tokens that do not meet the requirements, thus reducing the size of vocabulary and improving computing time. So we first compared the difference between a minimum frequency of 1 and 2. The results were clear, we minimized when we changed from no-minimum to two.

We decided to test different values for this parameter between 2 and 10 to see where we optimized this parameter. Then we also thought that the quasi-identical words (only different by the suffix *-s*) could be tested in the same phase of optimization and see how would react our loss functions to these two parameters.

The results are shown in figure 5.1, the blue line represents the accuracies when the model is not normalizing tokens with respect of *-s* forms, the orange line is when we used only normalized tokens (normalization was made towards the variant without an *-s* suffix.

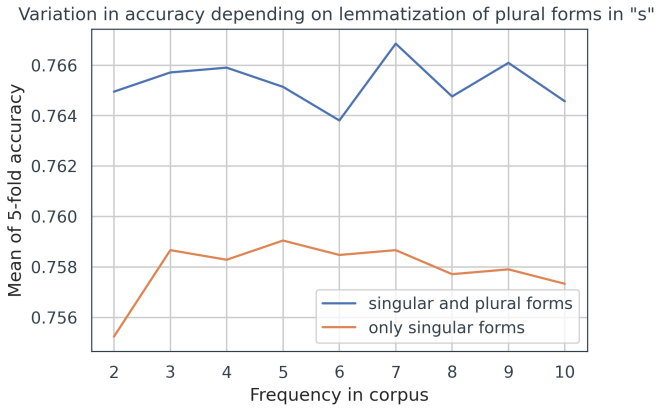

Figure 5.1: Normalization vs. No-Normalization

We observed that the blue line minimizes our loss function, so we know not to normalize with -*s*. Now, the next step was to check which value for minimum frequency worked best. For this test we also used the opportunity to try different values for smoothing (alpha). To see which configuration worked best, we chose the three best scores of accuracy from the blue line in the previous graph and tried different values of α between 0.001, 0.01, 0.1 and 1. The results are shown in figure 5.2

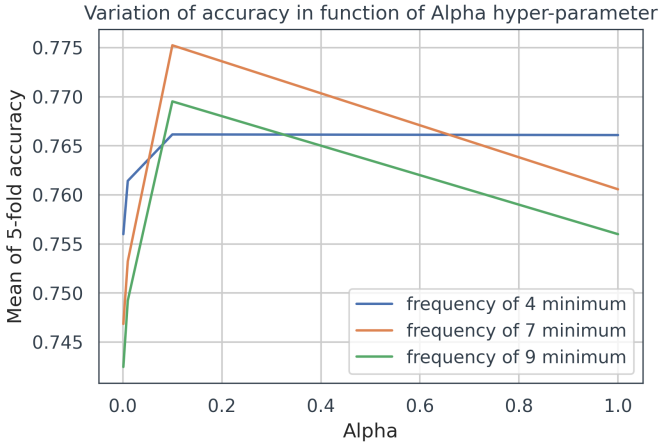

Figure 5.2: Values of alpha between 0.001, 0.01, 0.1 **and** 1

We focused on the area between the values of 0.1 and 1 because our loss function was minimized between these values of alpha. So we made incremental steps of +0.1, and from what we observe in the next graph, the minimum frequency of 4 and an α smoothing at 0.2 was the best accuracy. So That became our optimal parameters for our final Bernoulli NB model. This is shown by the next graph 5.3.

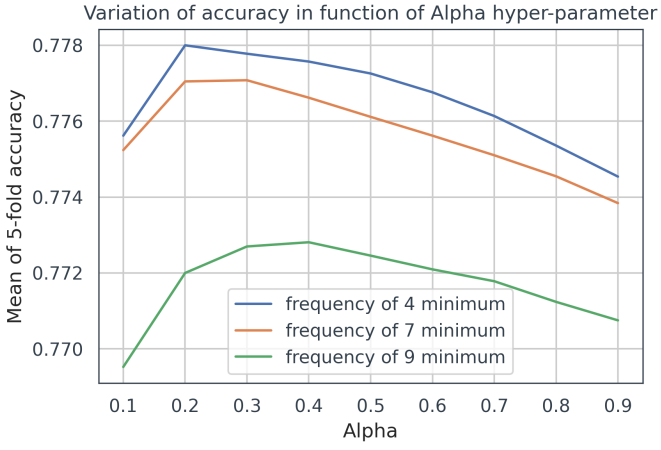

Figure 5.3: Values of alpha between 0.1 **and** 0.9

5.2 Multinomial NB

For the optimization of the Multinomial algorithm, we had to chose between type of vectors (BOW or TF-IDF), the range of n-grams, the minimum frequency to be considered a valid feature, and of course the α parameter.

We first started by dividing between BOW vectors and TF-IDF vectors and we used that opportunity to look for which values of alpha between 0.1 and 0.9 would improve accuracy. This is shown by figure 5.4

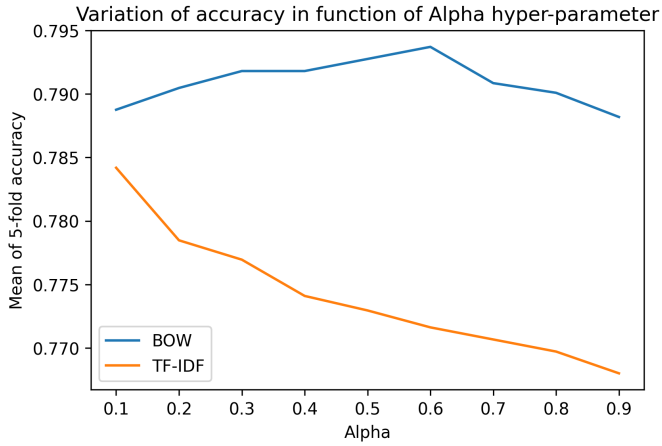

Figure 5.4: BOW vs. TF-IDF with respect to alpha

After the observing the data, we were surprised to see that BOW seemed better than TF-IDF, but our intuition told us to find parameters that might enhance TF-IDF's score. May be it is correlated with more context. So we tested both configurations but with ranges of n-grams from unigrams to bigrams. We also added the constraint of minimum 2 of frequency to reduce the process time and get rid of noisy tokens. The results are shown in figure 5.5

Just to be sure, we also ran the experiment with trigrams, but it lead to worst results, may be due to the curse of dimensionality or overfitting. Lastly, our last optimization for this algorithm was to find the threshold for minimum frequency in accordance with the smoothing hyper-parameter. We compared both methods for a value of alpha between 0.1 and 0.7. For the minimum frequency, we looked between values of 1 and 7.

After running both scripts, we compared the best results between BOW and TF-IDF and it seems that TD-IDF was the best with $\alpha = 0.2$ and minimum frequency of 1.

Accuracy
0.8019047619047619 alpha=0.2 min_df=1

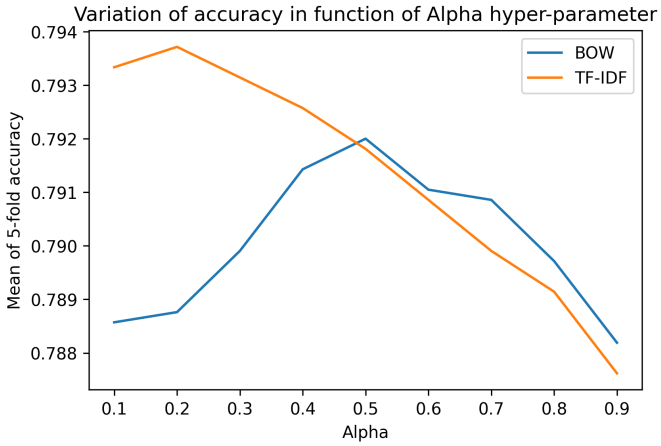

Figure 5.5: BOW vs. TF-IDF with respect to alpha

5.3 Support Vector Machine

The SVM was the last algorithm that we tested, so we had gained some knowledge from previous experience. We compared TF-IDF and BOW with the same default parameters, but this time it was clear that TF-IDF was the right choice. So we only tested SVM with TF-IDF transformation of data into vectors. We also used only unigrams and bigrams because it gave a little more flexibility than unigrams, but we did not want to overload our SVM with too many features considering the amount of time required to do one cross-validation.

The last parameters that we had to tune our model were : the minimum frequency, the type of kernel (linear, polynomial and gaussian (rbf)) and the size of the margin (C). We first ran a grid-search in order to findout if some parameters were right off the bat better than others. We tried all three kernels, different minimums of frequency (1 to 7) and 4 types of values for C between 0.001 and 10. From this experience, we observed that polynomial was doing poorly compared to other kernels and values of Kernel under 0.1 were not doing good either. So we needed a bigger margin and not a polynomial degree kernel. For example, the best value when C was equal to 0.001 = 10%, 0.01 = 7%, 0.1 = 60% and 1 = 78.8%. This observation makes sense considering that we are dealing with 15 different classes, we want to be more tolerant to induce bias but be better at generalizing.

Lastly, knowing that SVM can do well with a lot of features (not considering computing time), we compared different n-grams (between 1 to 3).

[3]We also tried a basic approach to KNN for this task, knowing that KNN are generally not great with manipulating a lot of dimensions. The result of this experiment were indeed pointing towards this. For different values of K between 1 and 7, accuracy was not higher than 35%

For this last experiment we tested all kinds of combination of parameters between C, minimum frequency, kernel and n-grams. We compared the best results for each type of n-grams range and it seems that the unigram to bigram combination was the best. Here are the results :

Vector Type	C	min freq.	kernel	n-gram range	score
TF-IDF	2.5	1	rbf	(1,1)	0.786
TF-IDF	7	3	rbf	(1,2)	0.791
TF-IDF	0.8	3	rbf	(1,3)	0.787

So we decided to use the best accuracy as an indicator for our best parameters and hyper-parameters for this algorithm. We were surprised to see that the size of the margin was this big compared to the others, but potentially after some point, the difference in size of the margin is not having as much as an impact as it was doing when we were comparing 0.001 and 1 for example.

6 Results

These results correspond to the mean of a 5-fold cross-validation on validation set. All three models were competitive enough, because there's only 2.3% accuracy between the best mode and the worst model. We think that the reason multinomial did better than Bernoulli is because frequency within a document was a feature in itself instead of just activating a switch if a feature is present or not. SVM came close second, but these algorithms are known for performing well on multi-class classification.

Model	Accuracy
NB Bernoulli	77.8
NB Multinomial	80.1
SVM	79.1

We will now look at the confusion matrices for all results, figures : 6.1,6.2, 6.3, and see where we did good and where the predictions got mis-labelled. This next section will start with the three confusion matrices that were generated from one of the 5 cross-validations. We will discuss the results after.

From looking at these confusion matrices we made some obersvations. First, they all had problems with the astro domain (astro.ph, astro.co, astro.sr, and astro.ga). That might be explained by the fact that they are so related in vocabulary that all models failed to predict the correct classes. That makes a lot of sense considering that we would expect this kind of behavior. On the one hand the SVM tries to linearly separate the data, while the Naive Bayes algorithm predict the class that is most probable depending on vocabulary. The other cluster of misclassification that we notice are the cond-mat that share a lot of mistakes on both sides. This is probably due to the same reason mentioned above. And lastly, we were surprised to see that cs.LG and stat.ML, which have

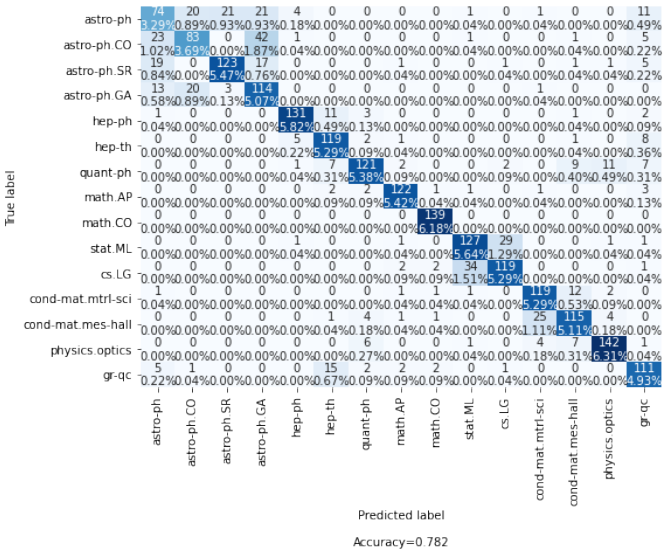

Figure 6.1: Confusion Matrix for 15 classes with Bernoulli NB

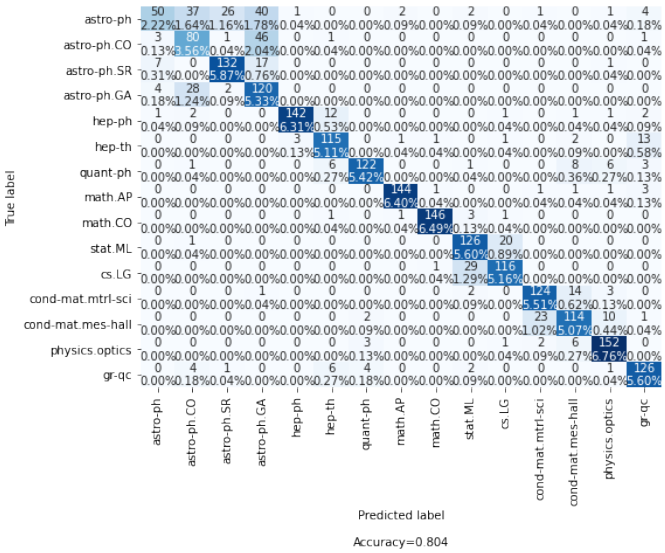

Figure 6.2: Confusion Matrix for 15 classes with Multinomial NB

different names, but are both labelled as *Machine Learning* on ArXiv's website. But surprisingly enough, that behaviour was not true for both hep and math. Those clusters ended up being well predicted in general, meaning that their features were probably different enough to distinguish them.

Overall, the performance of all three models were quite similar if we look at the shape and shades of the three confusion matrices that we generated. They made quite similar predictions.

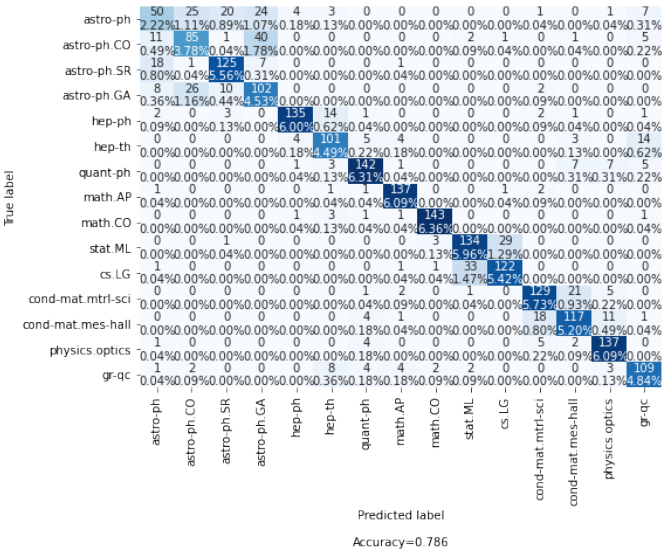

Figure 6.3: Confusion Matrix for 15 classes with SVM

7 Discussion

The pros of a Naive Bayes implementation was the efficiency in time. In between both approaches , Bernoulli and Multinomial, it was clear that generally Multinomial did better, and we think it's because of the non-binarization. Meaning that the difference in types of vectors helped. Also, the fact that we used TF-IDF also boosted our performances with giving better weights to meaningful features.

We think that the SVM did a good job also, but it took more time finding the best parameters considering that there are more parameters (type of kernel and the size of margin) and the processing time is much longer than a Naive Bayes approach. But it's important to note that it still did better than the Bernoulli approach, may be once again because the "on/off" switch of features was not the right answer to this problem.

Finally, an improvement to our task might have been to train word embeddings from a as much articles on ArXiv related to these classes. In this scenario, our model could have learned the representations in context and we might have been able to build a Doc2Vec system where each document is attributed a vector calculated from word embeddings that were created with the same kind of data. The classic Google News pre-trained embeddings is probably not fit for this task considering it's so news related and not really science oriented. It might also reduce the process because with word embedding we could fix a 300 number of dimensions instead of the thousand we are presently using for all of our models.

References

[1] Alfons Juan and Hermann Ney. Reversing and smoothing the multinomial naive bayes text classifier. In *PRIS*, pages 200–212, 2002.

[2] Pascal Vincent. Kernel trick - lecture notes in IFT6390. Universite de Montreal, October 2020.