Automatic Post Tagging

Theodoros Diamantidis, 112 and Sofia Kostoglou, 117

In this work, we used Pyspark as well as simple python to preprocess and train Machine Learning models for automatic post tagging of stackoverflow.com posts. The tags where limited among 'javascript', 'html', 'css' and 'jquery'. We provide a comparison in the performance of the ML models selected as well as a comparison between Pyspark and simple (sklearn) python for the data we were able to use. This work was completed within the context of the course "Mining From Massive Datasets" for the "Data and Web Science" M.Sc. program of Aristotle University of Thessaloniki.

I. INTRODUCTION

Text analytics, also known as text mining, encompasses techniques from various scientific domains such as artificial intelligence, statistics, and linguistics. Its applications span a broad range of fields, including security, marketing, information retrieval, and opinion mining. Unlike structured data that is readily amenable to analysis and evaluation, unstructured text necessitates transformation to reveal the underlying information it contains. Transforming unstructured text into a structured dataset is a non-trivial endeavor, and text analytics provides a diverse set of tools to handle the complexities, ambiguities, and irregularities inherent in natural language.

This work focuses on a text analysis problem involving feature extraction from text data, specifically addressing a multilabel classification task. In contrast to a multiclass task where each instance can only be assigned a single label from multiple options, multilabel classification allows for the assignment of multiple labels to each data instance.

For the multilabel classification task, we will utilize a subset of the "stackoverflow.com" data dump. This dataset consists of tsv files containing question and answer posts from stackoverflow.com. Each question on the platform is associated with several keywords, which we will consider as labels for the text present in the question's title and body. To construct our dataset, we have filtered the data to include only questions (including their title and main body) that are tagged with either 'javascript', 'css', 'jquery', or 'html'.

We are provided with the following data files:

1. **train.tsv:** This dataset serves as the ground truth data and contains four fields separated by tabs:

- Post ID
- Question title
- Full text of the question
- Set of labels assigned to the question/post, separated by commas.

2. **test.tsv:** This file has a similar structure to the training data but does not include the labels. The objective of this project is to assign labels to the questions in the test.tsv file, based on your solution developed using the train.tsv data.

Our first approach was to use Pyspark so we can achieve some better performance with the large amount of data we have. Unfortunately though, we ran into some technical problems and ended up running our experiments on Google Collab which has limited resources for non-subscribers. For that reason, we decided to also develop all our solutions in 'simple' python as well (i.e. sklearn, etc), in case we had better performance. We will present the results for both for the data sample we were able to use.

II. DATA PREPROCESSING

A. Data Cleaning

The first thing we want to do is clean our data so that our machine learning models can have better performance. Upon a quick look at the data, we see the following:

Each entry in the body column contains HTML elements like <p>, </p> for the beginning and ending of paragraphs, <a href="URL"> for hyperlinks, etc. It would be a good idea to remove these as much as possible so that we can keep the pure, clean body of each question in order for our tokenizer to work better.

Many entries include actual code that the OP has added to their question. It would probably be a good feature to know if we have code in a question, and even an estimation of what language it is based on some criteria. If not that, then just the information about having code or not in a question could be a feature since maybe some questions on certain languages can contain more code than other languages (e.g. people asking about Javascript may have code in their questions more frequently than people asking about HTML).

Since code in stack overflow is almost always included in specialized code snipets, we could potentially be able to

separate those snippets. After a quick google search, it appears as the <code> HTML tag is used for the piece of code, and this tag is usually included in the <pre> tag to tell the browser of google that the block of code contained is a block of code we want to display, not render.
After some analysis we decided to simply remove any code blocks from the body of the questions, as well as any HTML tags.

B. Feature Engineering

Having completed the cleaning step, we now proceed to feature extraction. Since we have text data, we use the typical preprocessing/feature extraction steps for NLP tasks, that is tokenization, removal of unwanted characters like exclamation points, commas, etc, removal of stop words and we also applied lemmatization on our tokens to get them to their 'base'/dictionary form. After this step, we applied tf-idf to extract features from our text data.

III. ALGORITHMS

In this section we will describe the machine learning algorithms we used to solve each task.

A. Problem Transformation Methods

The first method we used was to convert the multilabel problem into a multiclass. To do that, we used 3 different techniques:

1. *Binary Relevance*

Binary relevance is a popular approach used in multilabel classification tasks. In this approach, each label in a multilabel classification problem is treated independently as a separate binary classification task. Essentially, it decomposes the problem into multiple binary classification subproblems, one for each label.
To apply the binary relevance approach, the multilabel dataset is transformed into multiple binary datasets. Each binary dataset corresponds to a specific label and contains instances from the original dataset, but with the label of interest transformed into a binary target variable. For example, if there are four labels in the multilabel problem, four binary datasets will be created, each representing a separate label.
Once the binary datasets are prepared, a binary classification model is trained for each label independently. The goal of each model is to predict whether an instance belongs to the respective label or not. The models can be trained using various binary classification algorithms, such as logistic regression, decision trees, or support vector machines.
During prediction, each binary classifier is used to make a separate prediction for its corresponding label. This way, multiple binary predictions are generated, and the presence or absence of each label is determined independently. This approach allows for the possibility of an instance being assigned multiple labels simultaneously.
Binary relevance is a straightforward and intuitive method for multilabel classification, but it does not consider dependencies or correlations among labels. It assumes that the labels are mutually exclusive, which may not always hold true in real-world scenarios. Nonetheless, it serves as a useful baseline approach and can be combined with other techniques to improve multilabel classification performance.

2. *Label Powersets*

Label Powerset is an approach used in multilabel classification tasks that aims to consider the joint occurrences of multiple labels. Unlike the Binary Relevance approach, which treats each label independently as a separate binary classification task, Label Powerset treats the combinations of labels as distinct classes.
In the Label Powerset approach, each unique combination of labels present in the training data is treated as a separate class. For example, if there are three labels (A, B, C), and in the training data, instances are labeled with combinations like (A, B), (B, C), and (A, C), then these combinations will be considered as distinct classes.
To apply the Label Powerset approach, the multilabel dataset is transformed into a single-label dataset, where each instance is associated with a unique combination of labels. This transformation results in an expanded label space, where the number of distinct classes can increase exponentially with the number of labels.
During the training phase, a multi-class classification model is trained on the transformed dataset, where the objective is to predict the specific combination of labels for each instance. Various multi-class classification algorithms can be employed, such as decision trees, random forests, or neural networks.
During prediction, the trained model generates predictions for all possible label combinations. The predicted combination with the highest probability

or confidence is selected as the output for a given instance. This allows for the assignment of multiple labels simultaneously, taking into account the joint occurrences of labels.

Label Powerset considers the interdependencies among labels and captures the relationship between different label combinations. However, it can be computationally expensive and is susceptible to the curse of dimensionality when dealing with a large number of labels. Additionally, the label space can become extremely large, leading to potential data sparsity issues.

Overall, Label Powerset provides a more comprehensive and holistic approach to multilabel classification by considering the joint occurrences of labels, but it comes with trade-offs in terms of computational complexity and data requirements.

3. Classifier Chains

Classifier Chains is an approach used in multilabel classification tasks that takes into account the interdependencies among labels. It builds a chain of binary classifiers, where each classifier in the chain is trained to predict a specific label considering the predictions of previous classifiers in the chain as additional features.

To apply the Classifier Chains approach, the multilabel dataset is initially transformed into multiple binary datasets, one for each label. Each binary dataset contains the original features of the instances along with the labels that have been observed up to that point in the chain. For example, in a problem with three labels (A, B, C), the first binary dataset will include the original features, while the second binary dataset will include the original features and the predictions of the first classifier for label A, and the third binary dataset will include the original features, the predictions of the first classifier for label A, and the predictions of the second classifier for label B.

During the training phase, a binary classifier is trained for each label in the chain, taking into account the features from the previous classifiers as additional input. The order of the labels in the chain is typically predetermined or randomly chosen.

During prediction, the instances are passed through the chain of classifiers sequentially. The prediction for each label in the chain is based not only on the original features but also on the predictions of the preceding classifiers. This allows the model to capture label dependencies and exploit the correlations among labels.

Classifier Chains can handle label dependencies effectively, as each classifier has access to the predictions of the preceding classifiers. However, the order of the labels in the chain can affect the performance of the approach. Choosing a meaningful order that reflects the underlying dependencies among labels is crucial for achieving good results.

Classifier Chains provide a flexible framework for multilabel classification, allowing for the incorporation of label dependencies into the modeling process. It is computationally efficient compared to approaches like Label Powerset, as it trains binary classifiers independently. However, it may suffer from error propagation, where a mistake made by one classifier in the chain can affect subsequent predictions in the chain.

B. Approximate kNN Search

In this method, we first apply Locality Sensitive Hashing (LSH) to our data, and then for each instance we try to find its k approximate nearest neighbours. Then the tag(s) that appear with frequency more than some frequency threshold (which is a parameter of this method) is assigned to the new example.

Locality-Sensitive Hashing (LSH) is a technique used to efficiently approximate similarity search in high-dimensional spaces. It is particularly useful when traditional exact search methods become computationally expensive due to the curse of dimensionality.

The main idea behind LSH is to hash similar data points into the same or nearby buckets with high probability. This grouping of similar points allows for efficient retrieval of potential neighbors when searching for similar items. LSH achieves this by leveraging random projections or other proximity-preserving techniques.

C. k-Means Frequency Classification

In this method, we first applied the k-Means algorithm to cluster data into k clusters. Then, for each cluster, we found the tags that appear with frequency more than a frequency threshold (which is a parameter of this method) and assigned these tags to the cluster. Finally, for each new instance, we find the centroid that is closest to it and assign it the tags that correspond to the cluster of that centroid.

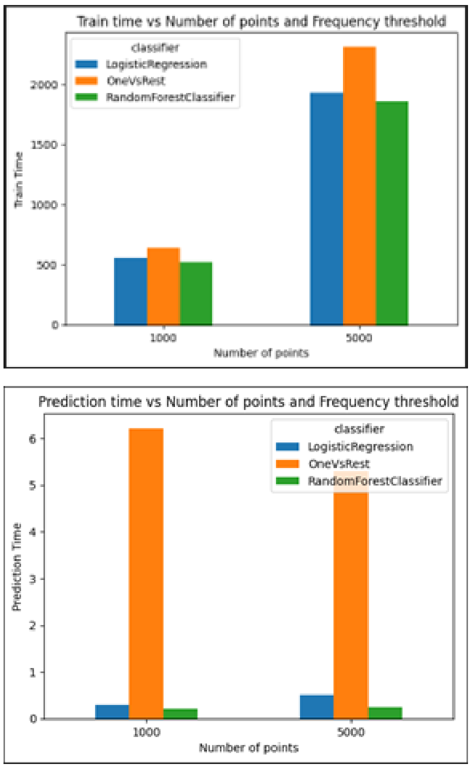


FIG. 1: Label powersets

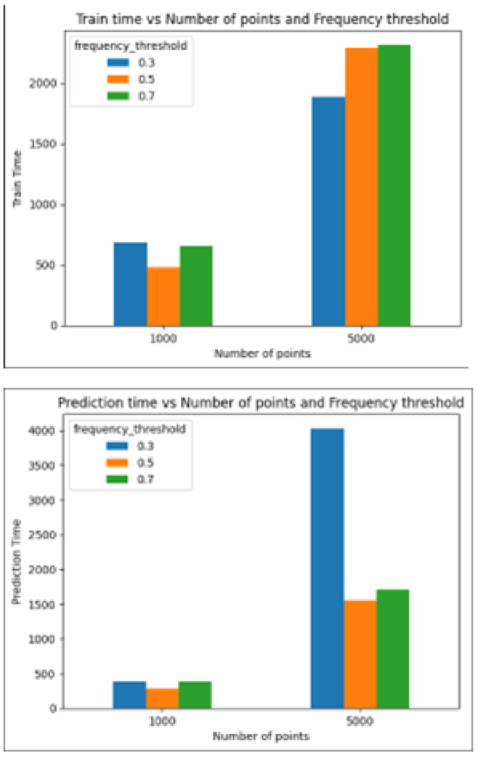


FIG. 3: kMeans

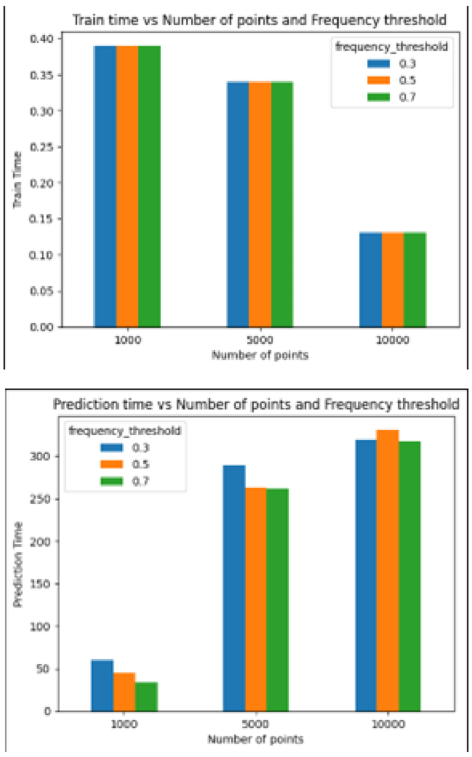


FIG. 2: kNN

IV. EXPERIMENTS

A. Setup

For Pyspark, we used Google Collab, but due to its limited resources we weren't able to achieve much parallelization from pyspark.

For the python implementation, we ran our experiments on a machine with 16GB of RAM and a 6 core CPU (AMD Ryzen 5).

As for our models' hyperparameters, we set the default values to be 0.5 for the frequency threshold (for both the Approximate kNN search and the k-Means Frequency Classification), 5 for the k in 'kNN' and 10 for k in k-Means.

Finally, the evaluation of our models was done using a variant of the F_1 score given by the following relationship:

$$F_1 = \frac{1}{N} \sum_{i=1}^{N} 2 \frac{|P_i \cap Y_i|}{|P_i| + |Y_i|}$$

where N is the number of instances, P_i is the set of predicted labels for instance i and Y_i is the set of real labels for instance i.

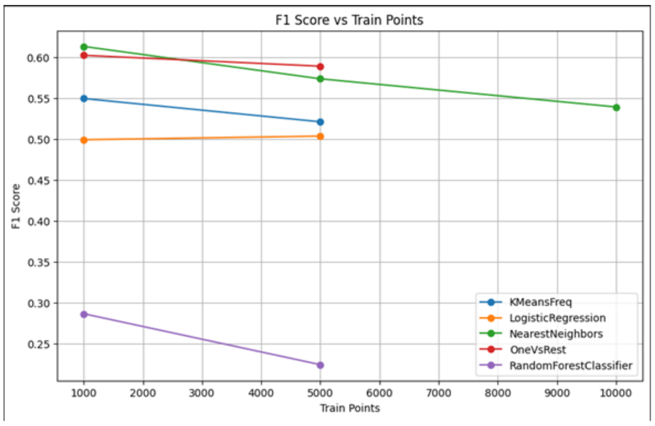


FIG. 4: F1 VS Number of points

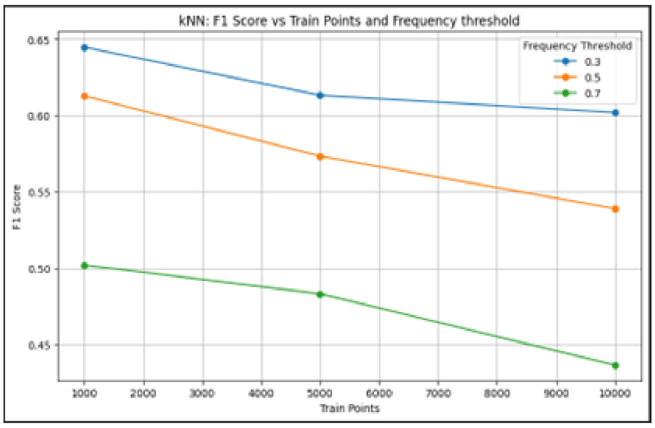
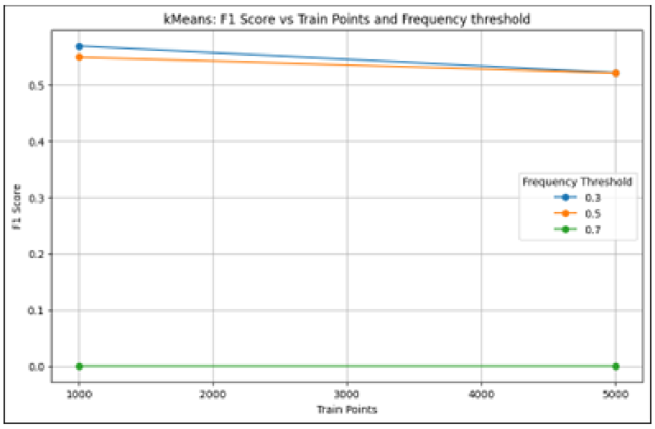


FIG. 5: kNN - F1 VS parameters



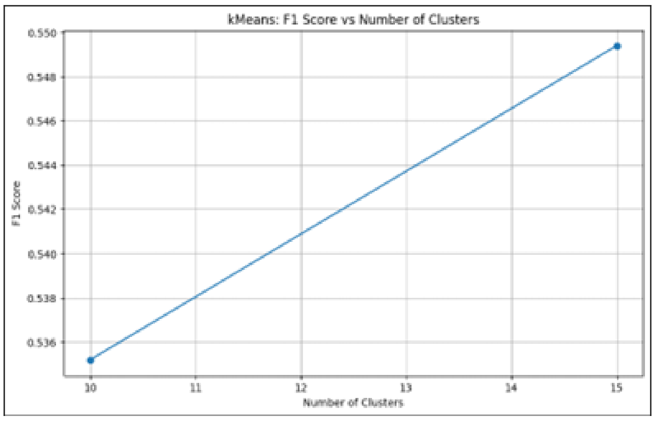


FIG. 6: kMeans - F1 VS parameters

B. Results - Pyspark

First, the different Pyspark algorithms are compared with respect to their efficiency and training/prediction times, for various parameters.

In figure 1, the first plot shows the training time of all 3 label powersets algorithms, LR, OneVSRest and RandomForest, for 1000, 5000 number of points. All three of them have similar training times for each different number of points, but we can surely see a dramatic change with the increasing of points. Regarding the prediction times, there are no differences at all, as prediction is pretty fast for all the algorithms.

In figure 2 the plot of kNN method is shown, for 1000, 5000, and 10000 points, compared to the frequency threshold that has been used. The training has constant timing, but for the prediction, the more the points, the bigger the prediction time. Especially, we spot the big scale up from 1000 to 5000 points, where 50 seconds become almost 300 seconds respectively.

Finally, in figure 3, we see kMeans method. Again, we spot big scale-up from 1000 to 5000 points, in both training and prediction time. Again, there is not a significant pattern to spot regarding the different thresholds used.

Then, in figure 4, comparison of all algorithms is plotted, having number of training points in axis-x and F1-score in axis y. The biggest scores belong to the OneVSRest and kNN classifiers, although the latter has a decreasing order as points increase. The RandomForest classifier seems to not be able to perform almost at all.

More interesting plots are shown in the two figures 5 above, for kNN and for kMeans respectively. Basically, the F1-scores of different frequency thresholds used are compared, having a threshold relatively small, at 0.3, being the best one for both the algorithms. Also, in figure , we see that as the number of clusters increasing in kMeans, so does the F1-score of the method.

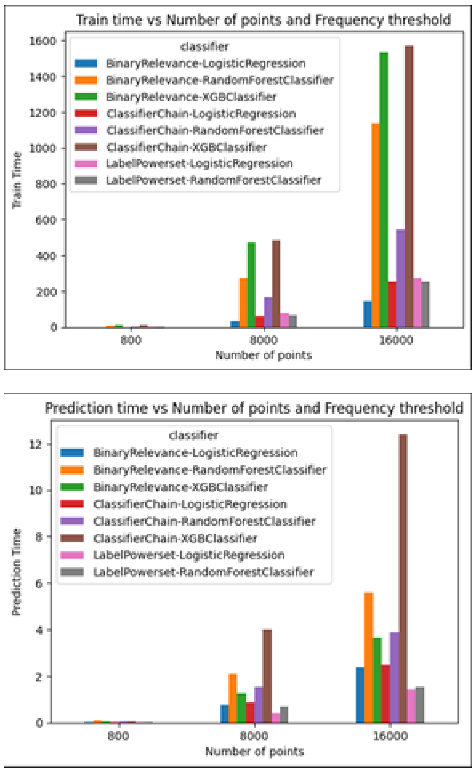


FIG. 7: Problem Transformation Methods

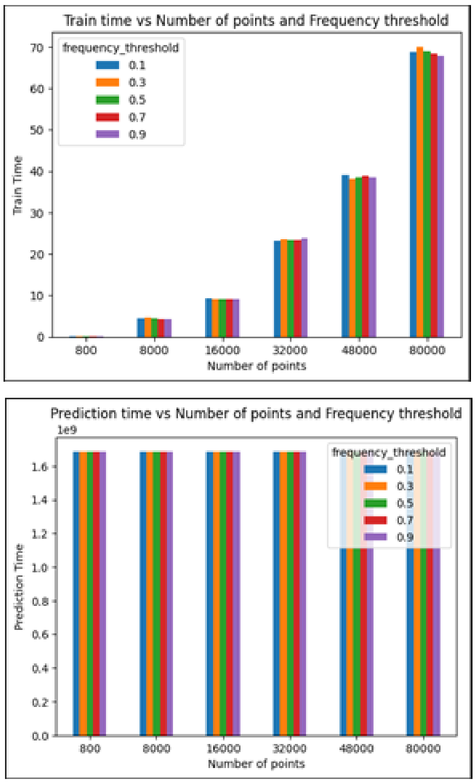


FIG. 9: kMeans

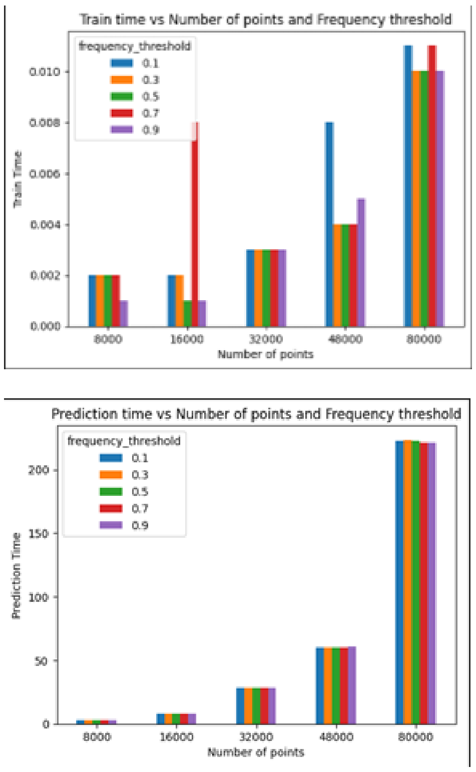


FIG. 8: kNN

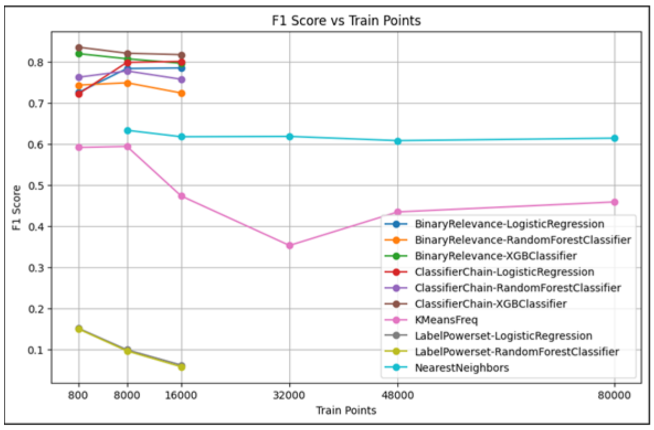


FIG. 10: F1 VS Number of points

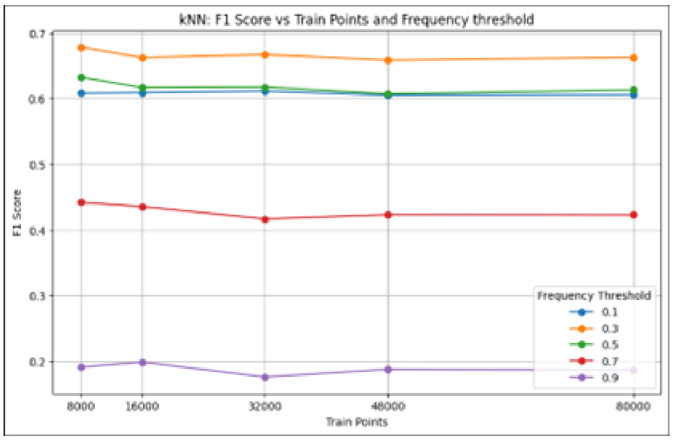


FIG. 11: kNN - F1 VS parameters

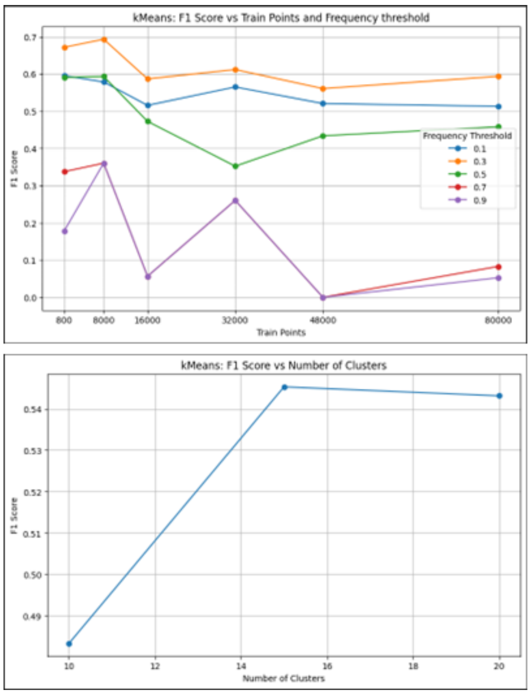


FIG. 12: kMeans - F1 VS parameters

C. Results - Python

Moving on to python implemented algorithms results, the same sets of plots are presented.
We start with showing the training and prediction times for all of the methods. First, in the figure 7, we start with problm transformation methods. As expected the train times increase rapidly, moving from 800 points to 16000, with XGBoost classifier being the most time-consuming among the others.

In figure 8, in kNN train time is negligible, as we discussed before, but there is a exponential growth in prediction time, with not much difference in the various frequency thresholds though.

In figure 9 , we see the same results for training time of kmeans, almost exponential again.

In figure 10, F1 scores are higher than the similar plot of pyspark implemented algorithms, as we see XGBoost perform at 0.8. Although, one can spot that as train points increase to 80000, most algorithms decrease their F1 scores, due to potential overfitting. But we can see in kMeans, that after some decrease in F1, form 50000 points and more, the F1 increases again.

Again, in the last two figures, kNN method and kMeans methods are plotted with respect to their number of points and F1 scores, to compare different frequency thresholds. The results are the same as discussed in pyspark, a relatively small threshold, not so small as 0.1 but around at 0.3, seems to have the best effect to our algorithms. For kMeans, the fact that the more clusters the better, remains true.

V. CONCLUSIONS

To sum up, we approached the problem with 2 different implementations, one with pyspark and one with plain python. Due to some technical reasons, we were not able to use pyspark's resources and optimize it well, to run our algorithms and experiments very fast. In contrast, in many cases python ran faster and for more points. Regardless, we were able to see how more complex algorithms for multi-class classification problems can be implemented in both.

So, in pyspark the OneVSRest method, which uses GradientBoosting binary classifier, achieved the best results, in terms of F1 score. In python, XGBoost with classifier chains has the best performance, with F1 scores reaching even 0.8. In general, the more the data points, the more the training and prediction time for the algorithms, as expected, having even exponential growth sometimes. In most cases, regarding the kNN and kMeans algorithms, the frequency threshold that was used, performed better when it was set at around 0.3, meaning we don't want a pretty small or a pretty large (strict) threshold. Finally, for kMeans, more clusters mean higher F1 scores, which makes sense as the classes that are produced for all combinations of tags in our problem, are many.