Machine Learning Project

Eva Battolla, Lima Madomi, Rachel Rieille

2023-12-22

Contents

Introduction

This project aims to process the Stack Overflow dataset using machine learning and address the challenge of identifying missing tags within posts. These tags serve as the essential thread connecting each post to its subject matter. The objective of this project is to accurately predict missing tags, using machine learning models and the nuanced application of Natural Language Processing (NLP). Besides NLP's giving machine the ability to process human languages it also has its own shortcomings, such as, unlike structured data, text data requires understanding of the co-context and semantics, since structured data patterns are more apparent. Additionally, training machines with this context and semantics is a challenge in itself. Words meanings can shift based on context, posing a substantial challenge. In today's world, there are multiple tools available to process languages, which help developers during their development of the models.

While working on this project we understood the importance of text data preprocessing and its impact on the model predictions. We realized that good preprocessing and tokenization could change the predictions result largely. Therefore, to meet the objectives of this project, we acquired a holistic approach. This included good preprocessing to clean and structure data and thoughtful feature engineering to reduce the noise and overfitting. Additionally, selecting the appropriate model and method that aligns seamlessly with this dataset. Challenges in this project made us understand the vast world of machine learning and its power in processing vast amounts of data.

Exploratory Data Analysis

Before diving into the overall coding and creating a machine-learning model that best aligns with our training and test dataset, we try to visualize and understand the dataset. This step of our approach assists us in better understanding the dataset and finding a model that best works with our dataset.

As a preliminary step for a quick insight into the dataset structure, column names, and the first few observations to make informed decisions about subsequent preprocessing steps, we reviewed the representation of the data in the dataset. The dataset contains three columns: 'Id', 'post' and 'tags' and 28'000 rows.

Table 1: First 5 rows of training data set

Id	post	tags
1	what is causing this behavior in our c# datetime type	c#
3	have dynamic html load as if it was in an iframe i have an asp.net 4.0 site. users can save an entire html page into the backend database. what i want to do is load the dynamic content into a div on an existing page in a content area and have a couple of things to happen: i do not want any of the css to affect anything outside the div when first trying this out loading of some badly formed html would move images and other divs outside the content area around. a lot of these html pages use the base tag for images and links i want the base tag respected inside the div. i have a solution that i am going to try which is just to use an iframe and set its url to another child page that loads the dynamic html into its own page entirely. i am just wondering if there is a better solution.	asp.net
4	how to convert a float value in to min:sec i m trying to convert my second in min:sec. my code is:	objective-c
5	.net framework 4 redistributable just wondering where we can get .net framework 4 beta redistributable. we would like to include it in our cd so we can distribute it to our clients and they need to install it from the cd and not from web as it is not necessary to have internet for our application. any suggestions will be appreciated. thanks navin	.net
6	trying to calculate and print the mean and its returning as a rather than a number python i have my program in python and i have used an external file with numbers in i created a list for the numbers to be stored in and then i need to find the mean standard deviation and the length of the list from this at the moment my program looks like this:	python

In the following visualisation, we aimed to see insights into the distribution of tags within the dataset. Using this method, we generated a bar plot that visualizes the counts of each unique tag in the 'tags' column. We use this plot to understand the prevalence of different tags and identify which tags occur more frequently than others. We also try to extract the unique values of tags. We aimed to use this plot to explore the tag distribution comprehensively and investigate the dataset's characteristics.

Table 2: Summary Statistics

Id	post	tags
Min. : 1	Length:28000	Length:28000
1st Qu.:10018	Class :character	Class :character
Median :19958	Mode :character	Mode :character
Mean :20006	NA	NA
3rd Qu.:30006	NA	NA
Max. :39999	NA	NA

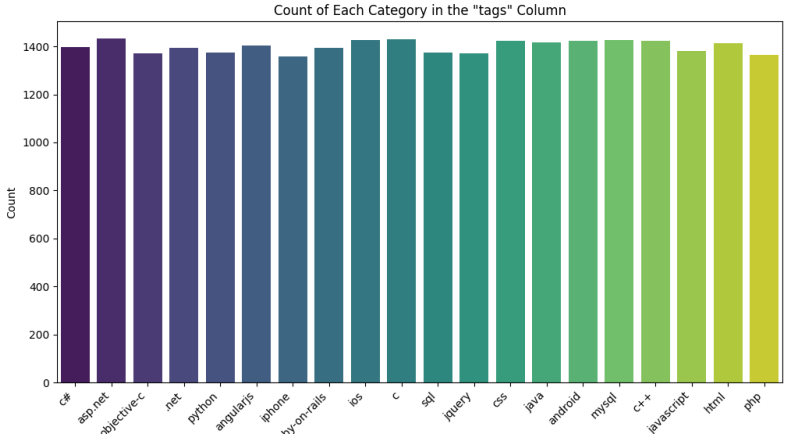

Figure 1: Frequency of tags in training set

Looking at these plots we discovered that in our dataset, tags are evenly distributed, with approximately an equal number of posts per tag. This balanced distribution helps mitigate the risk of class bias, allowing the model to learn more strongly. A balanced class distribution is beneficial for avoiding class imbalance biases, promoting better generalization to new data, and providing more meaningful evaluation metrics. It also helps prevent overfitting issues associated with imbalanced datasets.

Text processing

Examining the 'post' data reveals the necessity of text processing to retain pertinent information. As the posts come from a programming forum, the posts include raw html content which contains a lot of special characters. We first started with the following text-processing function. This function decodes html content, lowers every character, erases symbols due to HTML content, and erases stopwords to get the data to contain as few words as possible:

```
replace_space = re.compile('[/(){}\[\]\|@,;]')
bad_symbols = re.compile('[^0-9a-z +_]')
stopwords = set(stopwords.words('english'))
```

```
def preprocess_text(text):

    text = BeautifulSoup(text, "lxml").text
    text = text.lower()
    text = replace_space.sub(' ', text)
    text = bad_symbols.sub('', text)
    text = ' '.join(word for word in text.split() if word not in stopwords)
    return text
```

We realized through multiple tries that 'bad symbols' were potentially additional indications as the accuracy of the models improved when removing as few symbols as possible from the posts. We then decided to use lemmatization to minimize the number of different words, thus reduce de vocabulary and get a more accurate TF-IDF score. This score will be further explained later.

To address this, we employed a custom tokenizer function designed as follows based on the function found on the advanced topics in machine learning github [1]. The original function separated 'c#' in two different tokens. Since it is one of the possible tags, it was important that the model consider it as a single token. We added a few lines of code and ended up with the following solution :

```
def custom_tokenizer(text: str):

    text = text.replace('c#', 'csharpplaceholder')

    text_tokens = nltk.word_tokenize(text)
    output = []

    for w, pos in nltk.pos_tag(text_tokens, tagset="universal"):
        if w.lower() == 'csharpplaceholder':

            output.append('c#')

        else:
            if pos in ["VERB"]:
                pos = "v"
            elif pos in ["ADJ"]:
                pos = "a"
            elif pos in ["ADV"]:
                pos = "r"
            elif pos in ["NOUN"]:
                pos = "n"
            else:
                pos = "n"

            l = wnl.lemmatize(w, pos=pos)

            if l not in stopwords:
                output.append(l)

    return output
```

This versatile function tokenizes the text but also categorizes each token by its word type to enable the use of 'WordNetLemmatizer()' Lemmatizer and minimize the overall vocabulary as intended. By integrating these functionalities; we streamlined the text, focusing on its core components and reducing verbosity for more efficient data representation.

Below, part of tokenized text using our predefined function:

```
## ['[', "'cause", "'", ',', "'behavior", "'", ',', "'csharpplaceholder", "'", ',']
```

Once we proceeded to tokenize the text, we had a first overview of the vocabulary and length of the post. Below we can find the number of words per post.

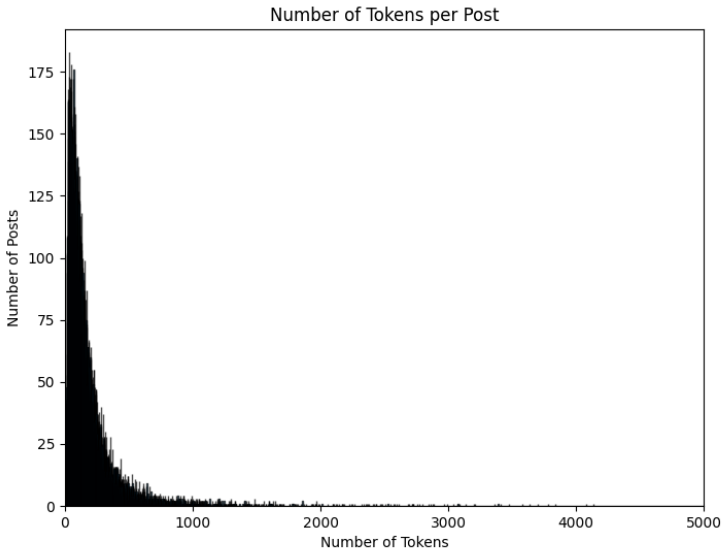

Figure 2: Number of Tokens per Post

According to Figure 2 above, we see that most posts have less than 5000 tokens. We can also look at the most used tokens per post. We observe that the use of special symbols is common to most of the tags but the frequency changes from one tag to another as as exhibited in the appendix.

Model Implementation and selection

To successfully tackle the classification task presented to us, we applied different machine learning models, ranging from Naive Bayes, k-Nearest Neighbors to other more complex models. We will focus on presenting some of the models, namely the Logistic regression, the Support Vector Machine (SVM) and the Random Forest model.

For each method, we used a Term-Frequency Inverse Document Frequency Transformer, which attributes to every token a weight relative to overall presence in document. Mathematically:

$$TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

$$\text{idf}_\text{i} = \log \frac{|D|}{|\{d_j : t_i \in d_j\}|}$$

The higher the TF-IDF score, the more important the term is to the document in the context of the entire corpus. The TF-IDF is fitted on the training corpus, we transform our training data and our test data with this TF-IDF values so that the data input in our model is no longer a list of character, but a matrix of

floating values, where rows are the different 'posts' and the columns are 'tokens' extract from the corpus. There are multiple parameters that can be set in the fitting of the "TfidfVectorizer" in Python that we used, for example the minimum value that a token needs to have to be kept in the fitted matrix. We set this value to '1e-3'. Once this transformation is done, the model will then learn to associate the values from the matrix of the TF-IDF passed on the training set, to the associated tags.

Logistic

[2]Logistic regression is a statistical method which is used for binary classification tasks to predict an instance belonging to a specific class. Conceptually, the model starts by forming a linear combination of input features. This model is used in machine learning as a supervised learning algorithm for binary classification tasks. Unlike linear regression, which predicts continuous outcomes, logistic regression models the probability of an instance belonging to the positive class using a logistic function, often called the sigmoid function.

Logistic regression in statistics and mathematics involves the log-odds transformation, likelihood function, and optimization techniques such as gradient descent. The log odds are modelled as a linear function of input features, and the model is trained by maximizing the likelihood of observing the given outcomes while minimizing the log loss, a measure of the difference between predicted probabilities and actual outcomes. Gradient descent is commonly used to adjust the weights iteratively, optimizing the model for accurate binary classification.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The regularization parameter (C) controls the trade-off between achieving a smooth decision boundary and accurately classifying training points. A smaller C encourages a wider margin, allowing for more misclassifications but enhanced generalization. Conversely, a larger C seeks a narrower margin, minimizing misclassifications at the cost of potentially overfitting to training data.

Our approach

Our primary objective in the logistic regression model was to construct and evaluate a tailored logistic regression model for identifying missing tags in the provided dataset posts.

As described above, our text preprocessing involved tokenization, lemmatization, and stopword removal using a custom tokenizer method. Subsequently we made a TF-IDF representations of the text. The logistic regression model was then trained using the training data, and its performance was evaluated on the test set.

GridSearch

To optimize the logistic regression model's performance, we used a grid search for hyperparameter tuning, namely, the hyperparameter 'C' previously presented.

Table 3: Grid Search for Parameter "C" in Logistic Regression

parameter	mean_test_score	std_test_score	rank_test_score
{'C': 10}	0.79520	0.00521	1
{'C': 1}	0.79107	0.00498	2
{'C': 100}	0.77536	0.00505	3
{'C': 1000}	0.76163	0.00555	4

We observe that the highest mean_test_score value was when the parameter 'C' set to fit the model was 10.

We then used the best parameters and the resulting model to make predictions on the test set. Finally, we created a confusion matrix to generate and display the model's classification performance further.

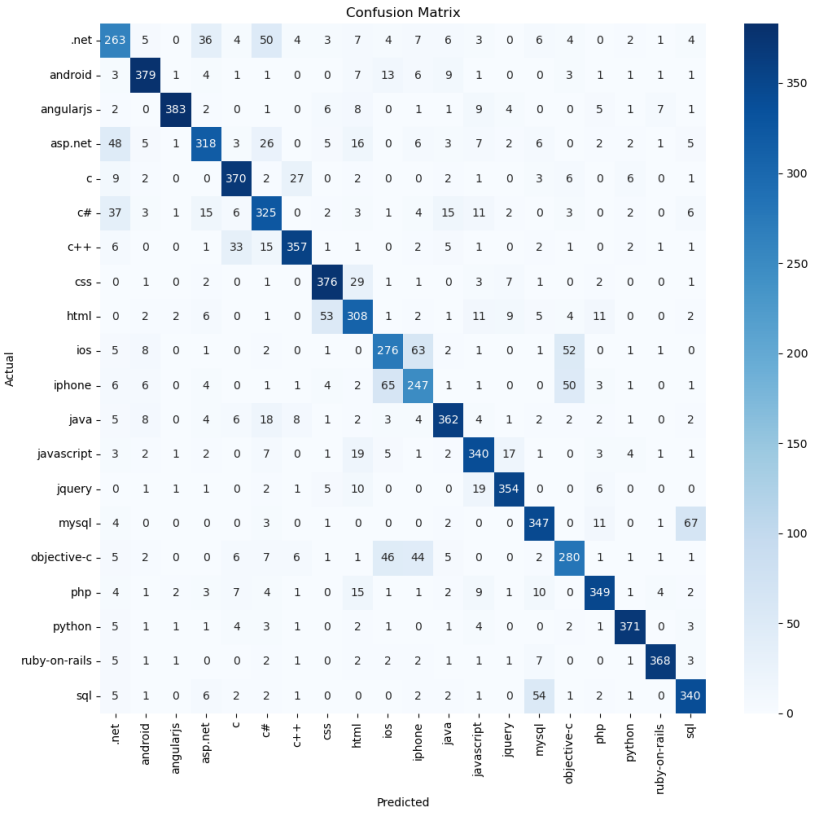

Figure 3: Confusion matrix for Logistic Regression

The confusion matrix shows how tags were misclassified. It seems that the models tend to misclassify 'mysql' and 'sql' but surprisingly not 'iphone' and 'ios' which was an issue that occurred with the first text processing technique we used. The accuracy per tag can be seen in the classification report table below.

Table 4: Classification Report on Training using Logistic Regression

Tags	precision	recall	f1-score	support
.net	0.6337349	0.6430318	0.6383495	409.0000000
android	0.8855140	0.8773148	0.8813953	432.0000000
angularjs	0.9720812	0.8886311	0.9284848	431.0000000
asp.net	0.7832512	0.6973684	0.7378190	456.0000000
c	0.8371041	0.8584687	0.8476518	431.0000000
c#	0.6871036	0.7454128	0.7150715	436.0000000
c++	0.8750000	0.8321678	0.8530466	429.0000000
css	0.8173913	0.8847059	0.8497175	425.0000000
html	0.7096774	0.7368421	0.7230047	418.0000000
ios	0.6587112	0.6666667	0.6626651	414.0000000
iphone	0.6284987	0.6284987	0.6284987	393.0000000
java	0.8578199	0.8321839	0.8448075	435.0000000

Tags	precision	recall	f1-score	support
javascript	0.7962529	0.8292683	0.8124253	410.0000000
jquery	0.8894472	0.8850000	0.8872180	400.0000000
mysql	0.7762864	0.7958716	0.7859570	436.0000000
objective-c	0.6862745	0.6845966	0.6854345	409.0000000
php	0.8746867	0.8369305	0.8553922	417.0000000
python	0.9321608	0.9251870	0.9286608	401.0000000
ruby-on-rails	0.9509044	0.9246231	0.9375796	398.0000000
sql	0.7692308	0.8095238	0.7888631	420.0000000
accuracy	0.7991667	0.7991667	0.7991667	0.7991667
macro avg	0.8010566	0.7991147	0.7996021	8400.0000000
weighted avg	0.8014256	0.7991667	0.7997963	8400.0000000

Despite our best efforts and multiple modifications to the model, we realized that this model was not a good fit for the dataset, and it did not perform with desirable results. The classification report exhibits an accuracy of 79.91% on the training set and obtained up to 91.14% on the test set on the Kaggle public Leaderboard. In an attempt to improve prediction accuracy, we explored other, more advanced models.

SVM

Support Vector Machines (SVM) is a powerful machine learning algorithm for classification and regression tasks. In the algorithm the primary focus is on creating a decision boundary that maximizes the margin between classes, defined as the distance between the hyperplane and the nearest data points. This characteristic allows SVM to generalize well to unseen data, especially in scenarios with complex decision boundaries. The versatility of SVM is important as it can handle both classification and regression tasks, making it applicable to various machine-learning scenarios. [3]

The regularization parameter C plays a crucial role in controlling the trade-off between smooth decision boundaries and accurate classification. In practical applications, SVM often leverages the kernel trick, accommodating non-linear relationships through various kernel functions. [4]

Our implementation involves using a Support Vector Machine (SVM) model with a radial basis function (RBF) kernel. The RBF kernel, being a real-valued function dependent on input and fixed points, shapes our decision configured as 'ovo' (one-vs-one). This configuration allows the model to build multiple binary classifiers, each distinguishing between two classes, with the final prediction relying on a voting mechanism.

Our SVM model was trained on the provided dataset. The training process was aimed to find the optimal decision boundary in the high-dimensional feature space. To adjust the minimized classification error, we used two crucial parameters, regularization parameters C and γ. For performance optimization, we conducted a grid search over a range of hyperparameters, leading to the selection of the best performing SVM model with ideal hyperparameters for subsequent testing and evaluation.

Tuning Parameters

In parameter tuning, the gamma parameter in the RBF kernel assumes a critical role, influencing the shape of the decision boundary. Smaller γ values yield a smoother boundary, while larger values create a more intricate boundary. The parameter 'C' has the same regularization function as in the Logistic regression previously mentioned.

GridSearch

In our code, GridSearchCV is employed to perform an exhaustive search over a specified hyperparameter grid, seeking the combination that maximizes the model's accuracy.

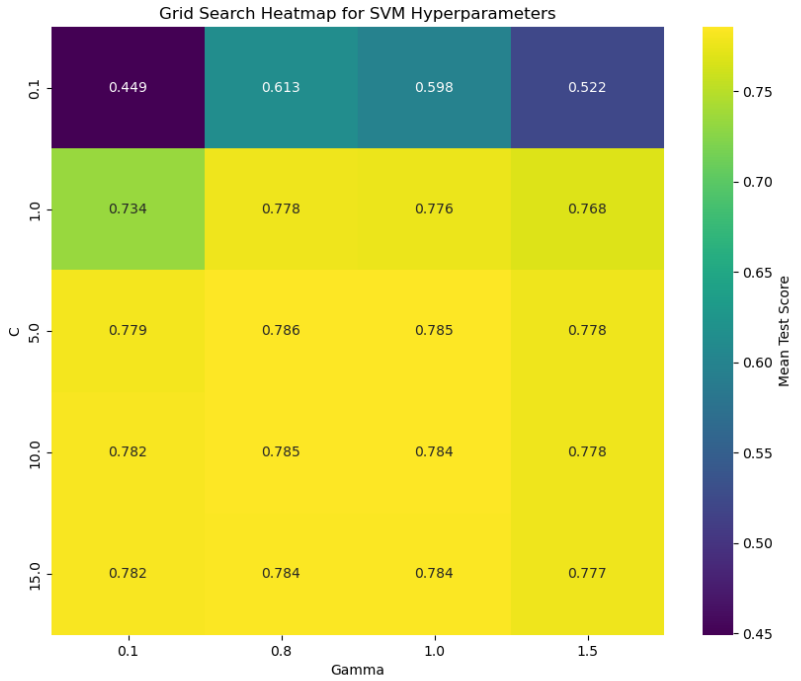

Figure 4: Heatmap from GridSearchCV Results

In our Gridsearch, we see that the optimal values for the tuning parameters are $\gamma = 0.8$ and C = 10.We also tried bigger parameters for C in a second GridSearch as 10 was our bigger proposed C, but the result did not change even with near values like 11.

Following the grid search, the trained SVM model was then applied to the testing data, to assess the using scikit-learn's 'accuracy score'. Additionally, we used a confusion matrix to visually represent true positive, true negative, false positive, and false negative predictions, helping in understanding the model's classification outcomes.

Table 5: Classification Report on Training using SVM Model

Tags	precision	recall	f1-score	support
.net	0.5899582	0.6894866	0.6358512	409.0000000
android	0.8924205	0.8449074	0.8680143	432.0000000
angularjs	0.9766234	0.8723898	0.9215686	431.0000000
asp.net	0.7894737	0.7236842	0.7551487	456.0000000
c	0.8344671	0.8538283	0.8440367	431.0000000
c#	0.6912752	0.7087156	0.6998867	436.0000000
c++	0.8775000	0.8181818	0.8468034	429.0000000
css	0.8449438	0.8847059	0.8643678	425.0000000
html	0.7078652	0.7535885	0.7300116	418.0000000
ios	0.6287016	0.6666667	0.6471278	414.0000000
iphone	0.6259542	0.6259542	0.6259542	393.0000000
java	0.8372642	0.8160920	0.8265425	435.0000000
javascript	0.7835294	0.8121951	0.7976048	410.0000000
jquery	0.9018088	0.8725000	0.8869123	400.0000000

Tags	precision	recall	f1-score	support
mysql	0.7881549	0.7935780	0.7908571	436.0000000
objective-c	0.6865672	0.6748166	0.6806412	409.0000000
php	0.8816121	0.8393285	0.8599509	417.0000000
python	0.9292929	0.9177057	0.9234630	401.0000000
ruby-on-rails	0.9527559	0.9120603	0.9319641	398.0000000
sql	0.7461024	0.7976190	0.7710012	420.0000000
accuracy	0.7939286	0.7939286	0.7939286	0.7939286
macro avg	0.7983135	0.7939002	0.7953854	8400.0000000
weighted avg	0.7988064	0.7939286	0.7956411	8400.0000000

The classification report indicates an accuracy score of 79.39% which is lower than the accuracy score exhibited by the Logistic regression. Surprisingly, it performed better on the whole data set as our Kaggle public leaderboard score with this regression obtained 93.761% accuracy. This result shows that the SVM model was better at learning new information from the whole training data set than the Logistic Regression, due to its complexity.

Despite achieving an optimal result on the Kaggle public leaderboard, a substantial improvement from the previous logistic regression, we further enhanced this result, and explored other models in subsequent steps.

Random Forest

Random Forest is a collective learning algorithm that is used to create a multitude of decision trees during training, and it outputs mode of the classes or the mean prediction of the individual trees. The key idea is to introduce randomness at both the data and feature levels. Each tree in the forest is built using a subset of the training data and a random subset of features at each split. This randomness enhances the model's ability to generalize well to unseen data and reduces overfitting.

From a statistical perspective, Random Forest incorporates diversity among trees, leading to improved generalization performance. It leverages the law of large numbers and decorrelates the individual trees by introducing randomness, reducing variance. The collective trees average out errors and tend to provide more stable and accurate predictions compared to individual trees. The algorithm is less sensitive to outliers and noise in the data, often requiring less hyperparameter tuning compared to other complex models.[5]

We developed our tailored Random Forest model and tuned it using Grid Search Cross-Validation in scikit-learn. The parameter grid (param_grid) was defined, specifying different values for key hyperparameters such as the number of trees in the forest (n_estimators), the maximum depth of the trees (max_depth), and the minimum number of samples required to split an internal node (min_samples_split).

The RandomForestClassifier was initialized with a fixed random state for reproducibility. Subsequently, we used GridSearchCV, taking the random forest model and the parameter grid. The grid search was conducted over a 5-fold cross-validation (cv=5), optimizing for accuracy (scoring='accuracy'). Finally, the best hyperparameters determined by the search were printed. This process helps identify the combination of hyperparameter values that maximizes the model's accuracy on the training data.

The best Random Forest model obtained from the grid search was then trained on the TF-IDF-transformed training data. We then generated key metrics, including accuracy and a classification report, to assess the model's performance on the test data. In this implementation, we trained, evaluated, and applied the Random Forest model for text classification, demonstrating steps from data preprocessing to model selection and prediction. The TF-IDF vectorization facilitates the conversion of textual information into a format suitable for machine learning, and the grid search aids in fine-tuning the model's hyperparameters for optimal performance. Using this model, we were able to achieve our best result, which was 0.94702. The initial results of this model were substantially low; however, by customizing the tokenizer and the word preprocessing, we were able to improve the result, making it our best model.

Tunning hyperparameter

Again, we used GridSearch to look for the best-fitting hyperparameters. We considered here three parameters:

First, 'n_estimator' corresponds to the 'number of trees in the forest'. More specifically the number of decision trees that are trained in the Random Forest. Increasing the number of trees generally improves the performance of the model, but it also increases the computational cost.

Secondly, 'max_depth' is the maximum depth of the individual trees. This parameter determines the maximum depth of each decision tree in the forest. A deeper tree can capture more complex patterns in the data, but it also increases the risk of overfitting. Setting it to None allows the tree to grow until it contains fewer than min_samples_split samples.

Finally, 'min_samples_split' mentioned above is the minimum number of samples required to split an internal node during the construction of a decision tree. If a node has fewer samples than min_samples_split, it will not be split. Increasing this value can prevent the model from becoming too complex and overfitting the training data. As this method has proved to be the best, we will investigate further the results of this model in the following section.

Table 6: GridSearch for parameters "max_depth", "min_samples_split", "n_estimators"

max_depth	min_samples_split	n_estimators	mean_test_score	rank_test_score
NA	10	500	0.80750	1
NA	5	500	0.80658	2
NA	10	600	0.80653	3
NA	5	600	0.80612	4
NA	20	500	0.80577	5
NA	20	600	0.80571	6
NA	10	300	0.80536	7
NA	20	300	0.80444	8
NA	5	300	0.80342	9
10	10	500	0.78061	10
10	20	500	0.78061	10
10	10	600	0.78041	12
10	5	500	0.78036	13
10	20	600	0.78020	14
10	5	600	0.78015	15
10	10	300	0.77872	16
10	5	300	0.77867	17
10	20	300	0.77786	18
5	5	600	0.76755	19
5	10	600	0.76679	20
5	5	500	0.76653	21
5	20	600	0.76617	22
5	20	500	0.76546	23
5	5	300	0.76515	24
5	10	500	0.76495	25
5	10	300	0.76429	26
5	20	300	0.76408	27

According to the GridSearch results, the best hyperparameters combination is the following: 'max_depth':None, 'min_samples_split':10, 'n_estimators':500, with a 80.75% mean accuracy score.

Using the confusion matrix with the best estimator, we tried to assess the performance of our model. This approach helped us identify the model's shortcomings on specific tags and helped us identify different approaches. The resulting plot is displayed, providing insights into the model's classification performance on the test data, and calculating the true and false positive. The confusion matrix was a valuable tool throughout our processing for understanding how well the model predicts different classes and identifying potential areas for improvement in its predictive capabilities. It was thanks to this analysis that we considered keeping 'c#' as a single token as it was often misclassified.

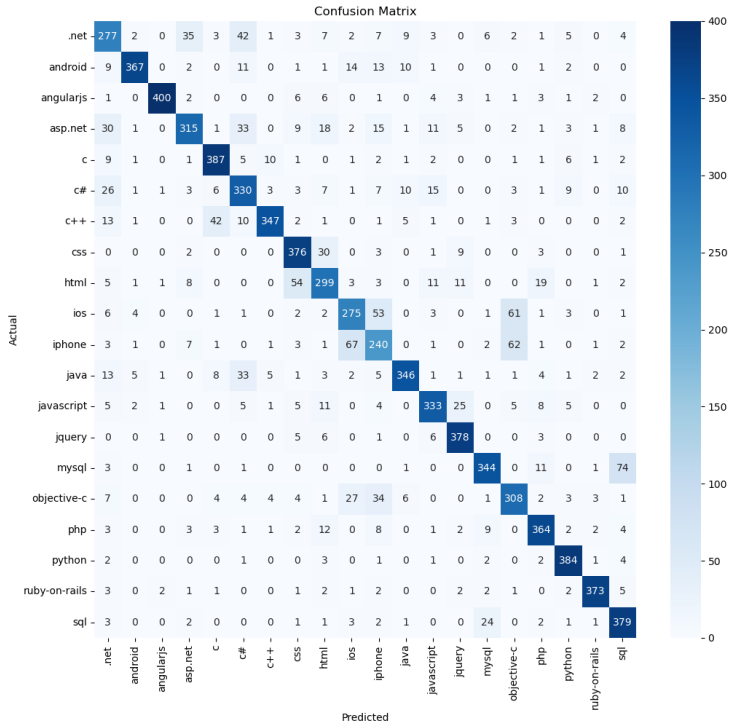

Figure 5: Confusion matrix for Random Forest

Table 7: Classification Report on Training using Random Forest Model

Tags	precision	recall	f1-score	support
.net	0.6626794	0.6772616	0.6698912	409.0000000
android	0.9507772	0.8495370	0.8973105	432.0000000
angularjs	0.9828010	0.9280742	0.9546539	431.0000000
asp.net	0.8246073	0.6907895	0.7517900	456.0000000
c	0.8468271	0.8979118	0.8716216	431.0000000
c#	0.6918239	0.7568807	0.7228916	436.0000000
c++	0.9302949	0.8088578	0.8653367	429.0000000
css	0.7849687	0.8847059	0.8318584	425.0000000
html	0.7274939	0.7153110	0.7213510	418.0000000
ios	0.6909548	0.6642512	0.6773399	414.0000000
iphone	0.5970149	0.6106870	0.6037736	393.0000000
java	0.8871795	0.7954023	0.8387879	435.0000000
javascript	0.8430380	0.8121951	0.8273292	410.0000000

Tags	precision	recall	f1-score	support
jquery	0.8669725	0.9450000	0.9043062	400.0000000
mysql	0.8730964	0.7889908	0.8289157	436.0000000
objective-c	0.6844444	0.7530562	0.7171129	409.0000000
php	0.8504673	0.8729017	0.8615385	417.0000000
python	0.8992974	0.9576060	0.9275362	401.0000000
ruby-on-rails	0.9588689	0.9371859	0.9479034	398.0000000
sql	0.7564870	0.9023810	0.8230185	420.0000000
accuracy	0.8121429	0.8121429	0.8121429	0.8121429
macro avg	0.8155047	0.8124493	0.8122133	8400.0000000
weighted avg	0.8165552	0.8121429	0.8125458	8400.0000000

Accuracy is now 81.2142% on our training data, which is the best we were able to achieve on the training data set only.

Results

While training the RF model with the training data we delved into feature engineering and parameter tuning to improve the results of the model prediction. The introduction of Grid Search Cross-Validation allowed us to systematically explore various combinations of hyperparameter values. Besides, improving the tokenizer and text processing had a significant impact on the improvement of the results. We were able to achieve 94.702% accuracy on the Kaggle public leaderboard, which is the best solution we were able to submit.

Predictions

Now that our best model gave us predictions, let's have a look at them and understand how the model predicted these. Taking 3 rows from the prediction file:

Table 8: Prediction file

Id	post	tags
47	rails: cannot set id for the what the object belongs_to during creation i have the following models but i am unable to set comment.post_id when creating it.	ruby-on-rails
50	event occur on scrolldown i am just trying to execute some code when the scroll bar of the page is about halfway down my page (or 500px). the issue is that this code works for going all the way to the bottom of the page to execute but if i supply numbers nothing happens when scrolling. does anyone know how to have it so the event will execute when scrolling 500px down works	jquery
51	what is the best way to change a base header and not redefine already defined variables background: i m working on a legacy project in c++. while extending the functionality of a base class(specifically i m adding multi-threading support) i had to modify a header file. let s call the base class header foobuilder.h . foobuilder.h defines a builder object that is used heavily later in the code. to implement multi-threading i included a bunch of windows specific headers. problem: in the other source files that include foobuilder.h(among other header files) i get error c2371: [windows variable name] : redefinition; errors during compilation. obviously if i could change every source file in my project to include foobuilder.h as the first header file i m good but there are a 101 files which include foobuilder.h! abstracting out the windows specific headers into stdafx.h is not an option. what is the best way to overcome this situation i m open to any and all suggestions. i really don t want to change the #include foobuilder.h position for every file!	c++

After some manipulation on the model object, using .predict_proba, we obtain the following probabilities :

Table 9: Probabilities of classification

Id	.net	android	angularjs	asp.net	c	c#	c++	css	html
47	0.00416	0.00669	0.01005	0.01162	0.02307	0.01182	0.01017	0.00390	0.00710
50	0.01742	0.02213	0.06665	0.03221	0.02045	0.03729	0.01415	0.01003	0.00839
51	0.03401	0.01888	0.00582	0.01363	0.02824	0.02376	0.65815	0.02506	0.01366

Table 10: Probabilities of classification

Id	ios	iphone	java	javascript	jquery	mysql	objective-c	php	python	ruby-on-rails
47	0.00704	0.01283	0.00540	0.00972	0.01277	0.00554	0.05341	0.01374	0.02283	0.75496
50	0.01221	0.00805	0.01460	0.08112	0.46179	0.01096	0.01610	0.08255	0.05854	0.01194
51	0.02671	0.01732	0.02234	0.00701	0.00344	0.00664	0.03525	0.01844	0.01984	0.01675

Probabilities exhibited in Table 9 and 10 show the model's degree of certainty regarding the assignment of a given sample to each individual class. During the training phase, each decision tree within the random forest learns distinctive patterns in the training data. Subsequently, when making predictions, each tree autonomously generates predictions for the class of a given sample. Higher probabilities indicate a higher level of confidence in the predicted class. The probability estimate for each class is the average of the predicted probabilities assigned to that class by all the individual decision trees in the random forest. The class with the highest probability estimate is then chosen as the final predicted class for the sample.

The formula is $P(c \mid x) = \frac{1}{T} \sum_{t=1}^{T} p_t(x, c)$.

The importance of features shown in Table 11 is calculated by measuring how the use of specific terms in the decision trees contributes to the reduction of impurity in the nodes. It appears that the most important features are usually tags name, which is not surprising. If a post contains the name of the tag, it is most likely that the post is tagged as this word. Most of the special characters do not appear in the table as they are used in most of the posts, as mentioned before, their presence raises the probability of selecting one tag more than another.

Indeed, the post with "Id" = 47 has been classified as "ruby-on-rails" with 75% certainty. In Table 7, the precision for this class was 0.9588689 on the training set, we can assume that post from this class provides specific tokens that are hardly present in posts from other classifications, as in Tables 4 and 5, the precision on this specific class was high as well. Seeing the word "rails" in Table 8 emphasizes this hypothesis. Looking at Table 11, we see that 'rail' is one of the most important features for the classification decision.

The post with "Id" = 50 has 46% certainty for the classification as "jquery". It is less than the previous certainty percentage. This can be explained by the fact that in the post, from Table 8, there are no highly scored "Important Feature". As for post with "Id" = 51, classified as "c++" with 65.815% certainty, we see that this higher percentage is probably due to the presence of "c++" in the post, but the importance score of this feature is less than 2 times the one from"rail", potentially due to the similarity of words like "objective-c","c++", "c#", "css", "c", if not correctly written in a post, could be tokenized as "c" and change the weight of these tokens.

Table 11: 30 Most Important Features After Fitting

Tokens	Importance	Tokens	Importance
android	0.0166212	c++	0.0085120
mysql	0.0163352	html	0.0082817
python	0.0147116	)	0.0079371
rail	0.0145807	angularjs	0.0079227
php	0.0145134	(	0.0077165
$	0.0143001	io	0.0077159
jquery	0.0140584	table	0.0076609
.net	0.0132763	c#	0.0076591
asp.net	0.0122999	angular	0.0073354
javascript	0.0119571	}	0.0070998
iphone	0.0111436	{	0.0070892
cs	0.0106129	printf	0.0070811
sql	0.0104095	:	0.0069828
java	0.0096622	var	0.0065077
;	0.0086701	function	0.0064577

Potential improvements and limitations

Adaboost

While trying to further improve our Random Forest model, we came across a potential candidate:

AdaBoost (Adaptive Boosting) is an ensemble learning algorithm that combines the predictions from multiple weak learners to create a strong learner. The basic idea behind AdaBoost is to sequentially train a series of weak learners on the same dataset, with each subsequent learner focusing more on the instances that the previous ones misclassified. The final prediction is then made by combining the weighted predictions of all the weak learners. AdaBoost has several hyperparameters that can be tuned to optimize its performance. The main ones are the n_estimators (the number of weak learners (e.g., decision trees) to train), the learning rate (a factor to shrink the contribution of each weak learner. Smaller values require more weak learners but may lead to better generalization) and the base_estimator being the type of weak learner used. It basically would use our previous RF model with the best parameters as a parameter inside our adaboost model.

Despite looking promising, we faced a big problem while trying to run the AdaBoost Grid Search as our computers continuously shut down. This did not allow us to find the best tuning parameters for fitting the model, thus leading us to abandoning this idea.

XGboost

After our failure with Adaboost, we further continued our search for a boosting model and found XGBoost, which is a Gradient Boosting Classifier. It is designed to improve upon traditional gradient boosting methods by incorporating regularization techniques and parallel processing. It sequentially builds a series of shallow decision trees, optimizing a regularized objective function through gradient descent. This method is widely used for classification tasks as it is known to be robust, efficient and effective. Sadly, we found this method too late, and the running time of this model didn't allow us to finish on time to get a result.

Conclusion

In conclusion, in this project we used Natural Language Processing (NLP) to identify the missing tags for the posts in the provided dataset. Achieved results were uploaded to Kaggle to a public leaderboard to compare all groups results. The challenge involved intricate nuances in natural language processing, as textual data introduces complexities beyond our initial expectation.

Our journey began with text preprocessing, employing techniques such as tokenization, lemmatization, and stopword removal to enhance the data's structure. The initial attempt using a logistic regression model showed limited success, prompting the exploration of more sophisticated models. The Support Vector Machine (SVM) model, equipped with a Radial Basis Function (RBF) kernel and configured as 'ovo' (one-vs-one), emerged as a promising approach. A meticulous training process involving key parameters like regularization (C) and gamma, coupled with grid search optimization, led to a well- performing SVM model. Despite achieving a commendable accuracy of 93.761%, the Kaggle leaderboard results prompted further exploration.

The Random Forest model, incorporating grid search and hyperparameter tuning, showcased its robustness in handling text data. Configured with optimal hyperparameters, it achieved our best result of 0.94702. The journey involved fine-tuning the TF-IDF vectorization and customizing the tokenizer to enhance predictive accuracy.

Overall, this project unveiled the power of machine learning in processing extensive datasets and navigating the intricacies of language. Each model iteration contributed to a deeper understanding of the challenges posed by unstructured textual data, reinforcing the importance of thoughtful preprocessing and model selection. The project's evolution reflects a continual pursuit of improvement, underscoring the iterative nature of data science and machine learning endeavors.

References

1. Engelke, S.: NLP normalization and TF-IDF: Seminar 06, (2023)
2. Friedman, R.T.T.H.J.H.: The elements of statistical learning. Springer (2009)
3. Kowalczyk, A.: Support vector machines succinctly. Syncfusion, Inc (2017)
4. S. Guido, A.C.M. &: Introduction to machine learning with python. O'Reilly Media, Inc (2016)
5. Breiman, L.: Random forests. Machine learning, Kluwer Academic Publishers. 45, 5–32 (2001)
6. OpenAI: ChatGPT - language model, (2022)

Appendix

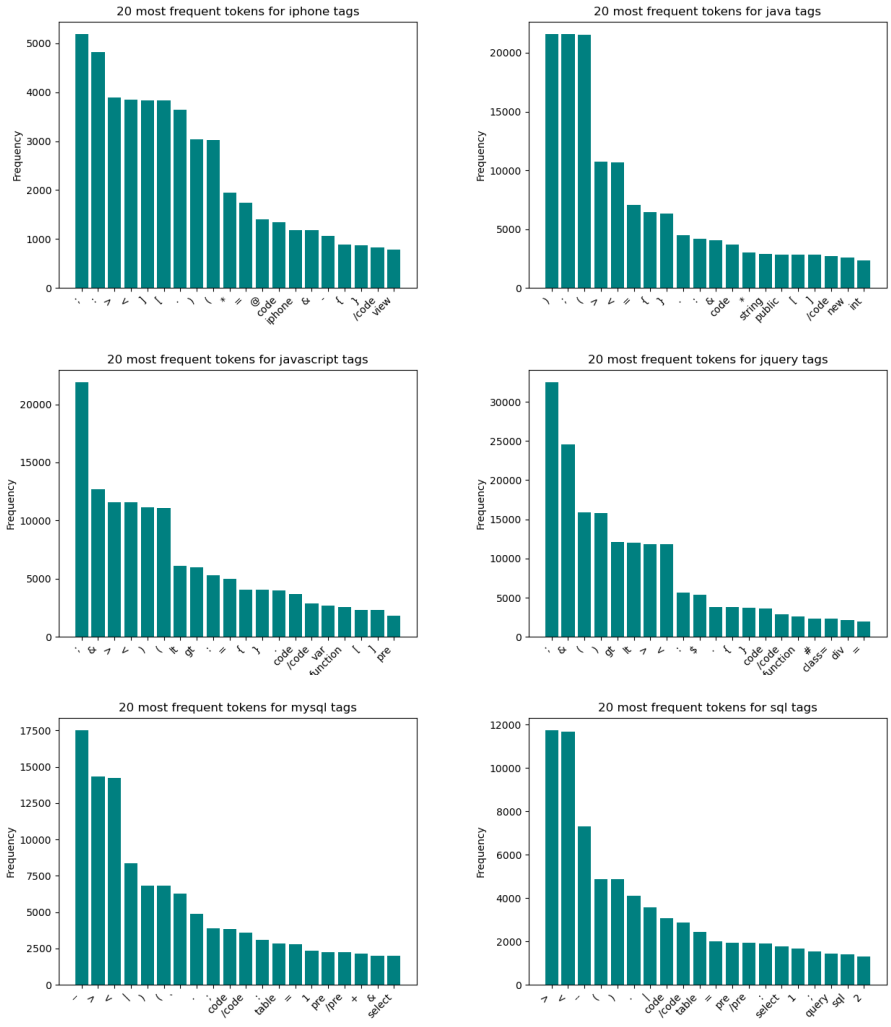

Figure 6: Most Used Words by Tag 1-6

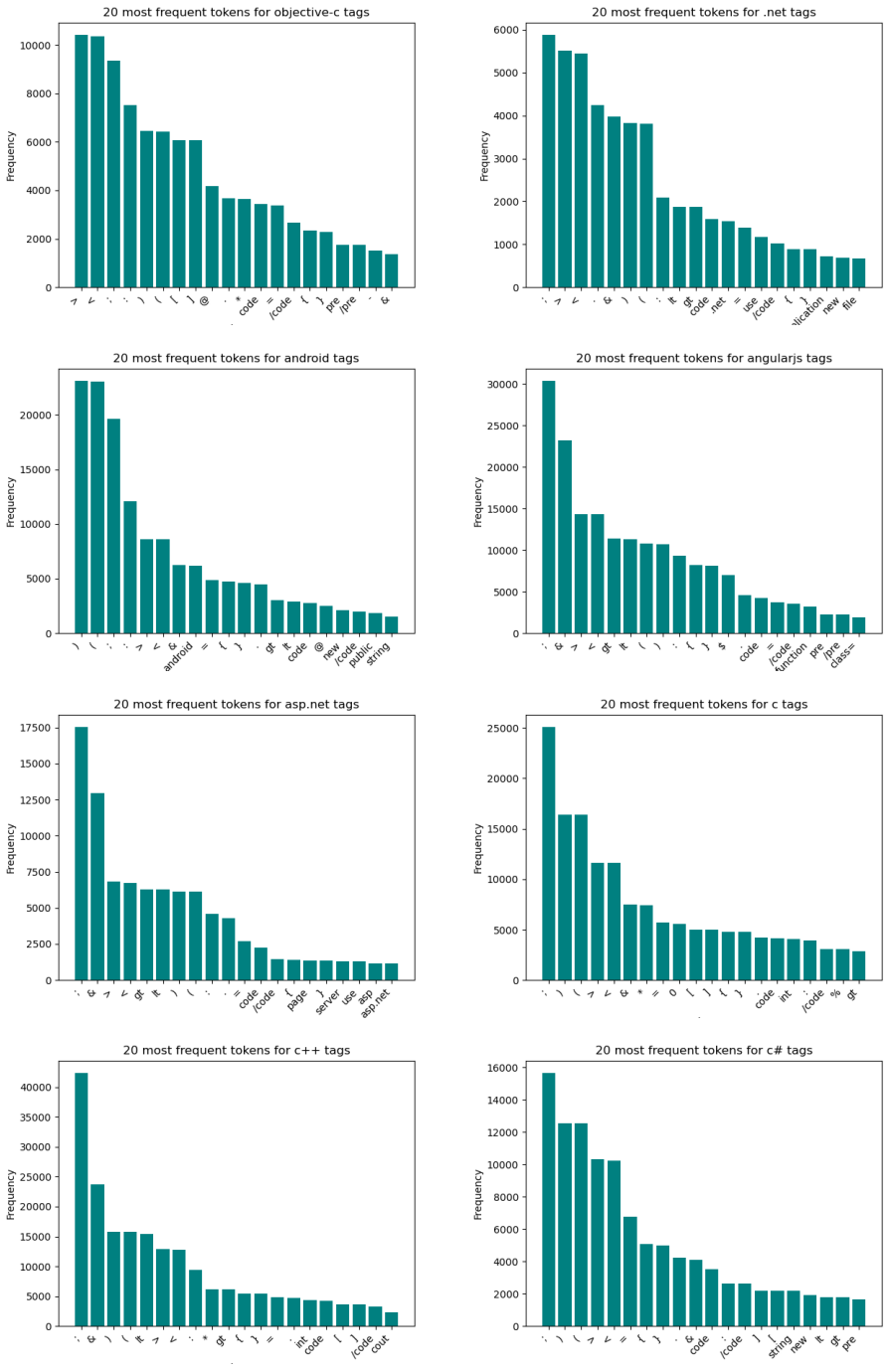

Figure 7: Most Used Words by Tag 7-14

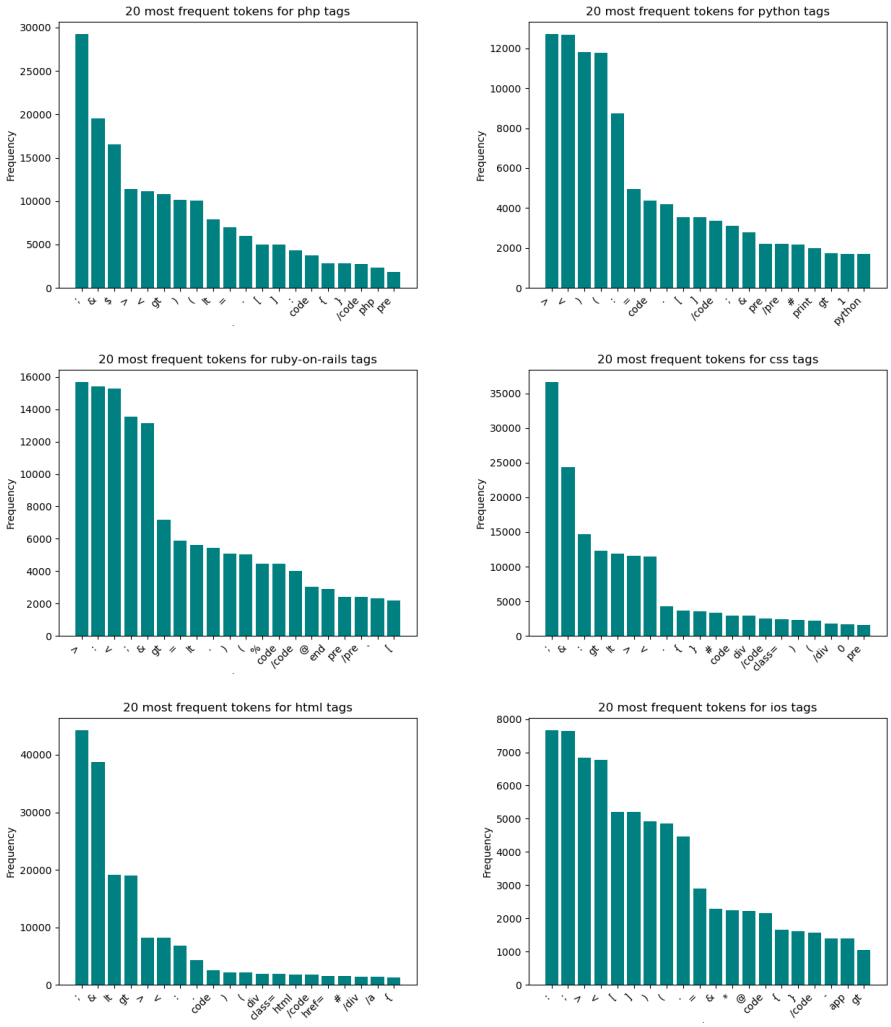

Figure 8: Most Used Words by Tag 15-20