UNIVERSITÉ DE GENÈVE
GENEVA SCHOOL OF ECONOMICS AND MANAGEMENT

Master in Business Analytics

Prof : Sebastian ENGELKE

MACHINE LEARNING PROJECT

Basics of natural language processing

Prepared By :

Oscar Wieland
Gentrit Alija
Pablo Huber

Contents

1 Introduction **1**

2 Description of the data set and exploratory analysis **2**

2.1 Description of the data set 2

2.1.1 Class Distribution of the Dataset 2

2.1.2 Words Distribution in Classes 2

2.2 Essence of the different Classes 3

2.3 Introduction to vectorization and tokenization 4

3 Selection and description of predictive models **5**

3.1 Selection of predictive models 5

3.2 Description of predictive models 6

3.2.1 Multinomial logistic regression 6

3.2.2 Support Vectors Classifier 8

4 Improvement of the model **10**

4.1 Choice of models features 10

4.1.1 Amelioration of CountVectorizer 10

4.1.2 TF-IDF 11

4.1.3 Lasso and Ridge penalty 11

4.1.4 Max iter and tol 13

4.1.5 Class_weight 13

4.1.6 Special parameters of multinomial logistic function 14

4.1.7 Special parameters of linear SVC 14

4.2 Enhancing Dataset Processing 15

4.2.1 Enhancing text formatting 15

4.3 Augmenting the dataset 17

5 Results **18**

5.1 Table of results 20

5.2 Confusion matrices and Classification report 21

6 Conclusion **24**

7 Bibliography **25**

8 Annexe **26**

1 Introduction

The purpose of this project is to create a highly accurate machine learning model that Stack Overflow may utilize to automate the process of single-tagging user-generated articles. These tags facilitate efficient information retrieval and contribute significantly to the platform's usability, allowing users to filter and access content pertinent to their interests and expertise levels. Our approach is designed to anticipate these tags by assessing the content and structure of the text.

The project will start with a comprehensive exploratory data analysis that looks at the distributions and patterns in the datasets, which includes the postings and the tags that go with them. This phase is critical for understanding the structure and nuances of the data we aim to model.

Then we will try some models seen in the lecture without any specified parameters and select 2 of them based on their training time, test time and accuracy. Subsequently, we will explain the maths behind those models and the differents features they can utilize. Next, we will explore the field of feature engineering, which transforms unprocessed textual input into a format appropriate for machine learning systems. This includes applying methods like vectorization, tokenization, and term frequency-inverse document frequency (TF-IDF) to highlight the significance of particular terms in the dataset.

Finally, we will evaluate the performance of our machine learning models for tag prediction on Stack Overflow employing a measure of accuracy defined by the proportion of correct predictions out of all predictions made, as per the standard formula:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

This statistic measures the frequency with which the model correctly recognizes the tags associated with each post, providing a simple way to assess the prediction effectiveness of our model.

2 Description of the data set and exploratory analysis

2.1 Description of the data set

2.1.1 Class Distribution of the Dataset

Ensuring that the Classes distribution is balanced will help in feature optimization later on, as some parameters are made for models using balanced dataset :

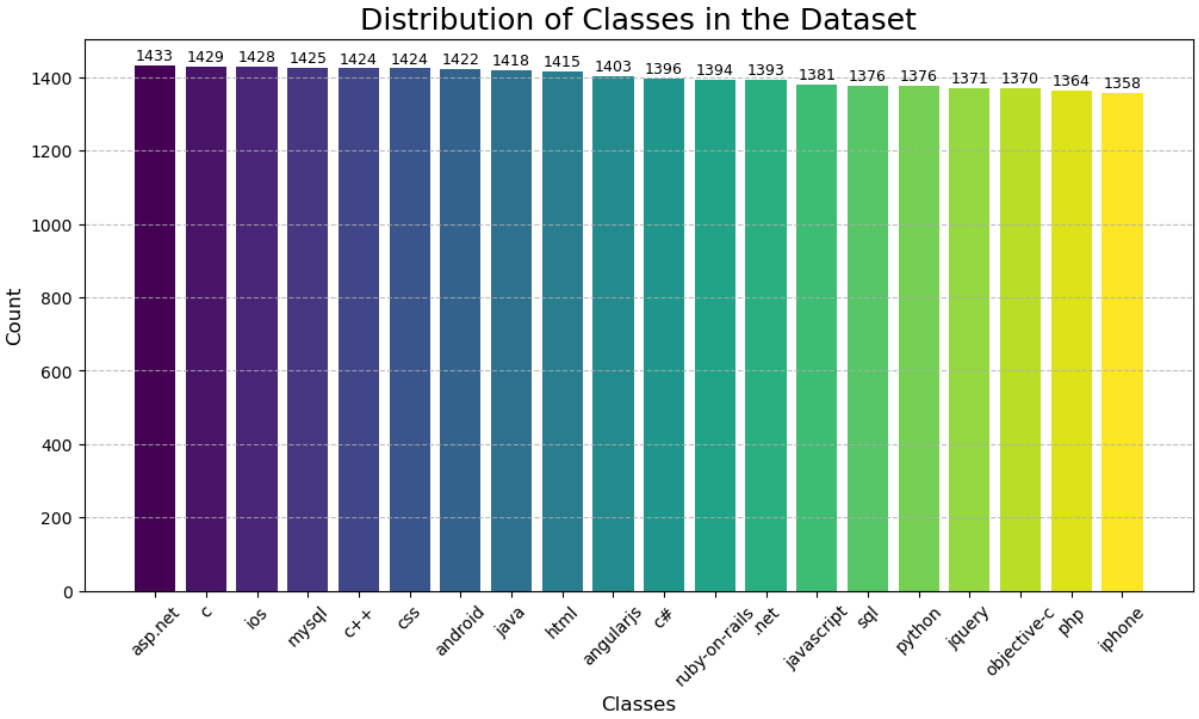

Figure 1: Distribution of classes in the dataset

The graph displays the class distribution of tags in the dataset. It shows a relatively uniform distribution, indicating that the dataset is well balanced. This equitable distribution of classes will likely contribute to more reliable and unbiased predictive performance when the dataset is used to train classification models.

2.1.2 Words Distribution in Classes

We then thought it might be interesting to have a deeper analysis of the tags and to analyze which words are the most common in each class. Thus, we produced a wordcloud for each class representing the words that are most present in each class with their frequency. This is very helpful to compare the important words of each tag and see if any of them are in several classes. For a concern of clarity, we decided to produce this graph after the first simple text formatting. It allowed us to view the important words without stop-words.

Figure 2: wordcloud for tags ios, iphone and jquery

The complete wordcloud table is in the annexe.

2.2 Essence of the different Classes

According to Stackoverflow guidelines, tags are concise descriptors that capture the substance of a question. Each time that a post is published, it needs to contain one or several tags that will help connect the experts on the matter with questions they will be able to answer.

Ideally, users should know and respect those specific tags guidelines. However, it is probable that some people won't take the time to look or implement those guidelines, leading to posts being mislabeled.

For example, the guideline for the `[iphone]` tag is:

> "DO NOT use this tag unless you are addressing Apple's iPhone and/or iPod touch specifically. For questions not dependent on hardware, use the tag `[ios]`. More tags to consider are `[xcode]` (but only if the question is about the IDE itself), `[swift]`, `[objective-c]`, or `[cocoa-touch]` (but not `[cocoa]`). Please refrain from questions regarding the iTunes App Store or about iTunes Connect. If using C#, tag with `[mono]`."

While the `[ios]` tag guideline is:

> "OS is the mobile operating system running on the Apple iPhone, iPod touch, and iPad. Use this tag `[ios]` for questions related to programming on the iOS platform. Use the related tags `[objective-c]` and `[swift]` for issues specific to those programming languages."

But after inspecting a few posts, we found an `[iphone]` labeled post that is mislabeled in our Dataset (ID = 62):

> "displaying addressbook data according to a certain search criteria. I need to display `abpeoplepickernavigationcontroller` according to a certain search criteria. I pass a name from my view. I need to display the `abpeoplepickernavigationcontroller` with that name only. How can I do this? Thanks in advance!!!"

It is labeled as `[iphone]` but doesn't respect the iPhone guidelines as it is not dependent on the hardware! Instead, it should have been labeled `[ios]` as it's related to "programming on the iOS platform".

We believe that this kind of miss-classification could be one of the reasons that closely related tags are hard to differentiate by our models and could explain why top accuracy scores of the competition don't seem to be able to surpass the 95% score.

Another reason for miss-classifications might be that, for instance, related or miss-classified tags contain similar words. If we take a closer look at `[iphone]` and `[ios]` respective posts' content, using word clouds (Figure 2), we can notice that both contain "app" and "code" as some of the most common words. Therefore, our models will have some difficulties distinguishing between the two classes. We can see this problem in the confusion matrix (cf.result part), which shows that `[iphone]` is often predicted as `[ios]` and vise versa.

To clarify how a model assigns a tag, we will show in the following lines how our model was influenced by the words present in a correctly labeled jQuery post to predict it:

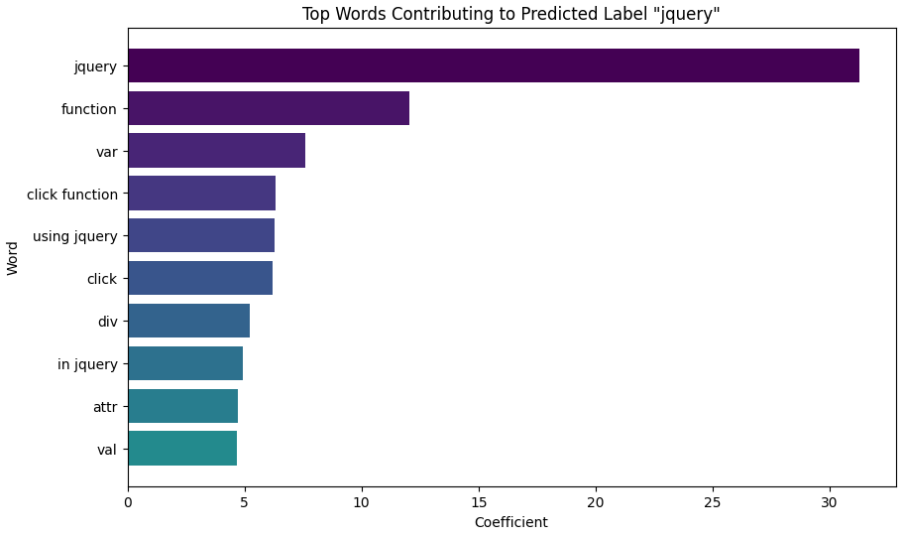

Figure 3: Top words for jquery

The coefficients next to each word are the coefficients associated with each feature present in the jquery post. They express the strength of the correlation between the feature made of a word or 2 adjacent words and the predicted label. Here we can see that the word jquery accounts for a lot in the selection of the label jquery, compared to the word "val" for example, which is approximately 6 times less significant.

2.3 Introduction to vectorization and tokenization

To create and use the model we studied, we first had to convert the text input (posts) as numerical data. This step is often referred to as "vectorization". In fact, machine learning algorithms typically operate on numerical data. They work with numerical input features in order to perform mathematical operations to learn patterns and relationships. This is computationally more efficient.

To do so, we used one of the simplest models for natural language preprocessing in machine learning: Bag-of-words. It works by separating the text into tokens (words) and counting the number of times each word appears in the text. For example, if the post is “i am studying for machine learning so i am tired”. Bag-of-words will create something as:

i	am	studying	for	machine	learning	so	tired
2	2	1	1	1	1	1	1

Table 1: Example of Bag-of-words

Now, the text is represented by the frequency of each word. In Python, this can be done with *CountVectorizer* from the package *sklearn*. CountVectorizer will create a matrix of token counts where the number of features is equal to the vocabulary size. Thus, in our case, *CountVectorizer* will create a matrix with a large number of columns that correspond to the number of different tokens in all the training sets and the rows will corresponds to the posts. Unfortunately, as we have seen in our accuracy results, using simply *CountVectorizer* may not be sufficient.

3 Selection and description of predictive models

3.1 Selection of predictive models

To narrow our focus, we only tested models that we have seen during the course and we made an Accuracy-time trade-off plot to help us decide . The reason being that some models don’t fit a certain type of dataset and therefore, will give bad results. For instance, as shown in the plot, KNN models behave poorly due to the fact that our dataset is a sparse matrix and KNN models are known to suffer from "the curse of dimensionality". Additionally, we expected some models to take more time to train and test than others. With a limited computer power and a project deadline, time demanding models would be hard to optimize as doing a Grid Search would take too much time.

In consequence, the models that we will optimize are the ones that gives a good result regarding their respective training + test time and test accuracy score.

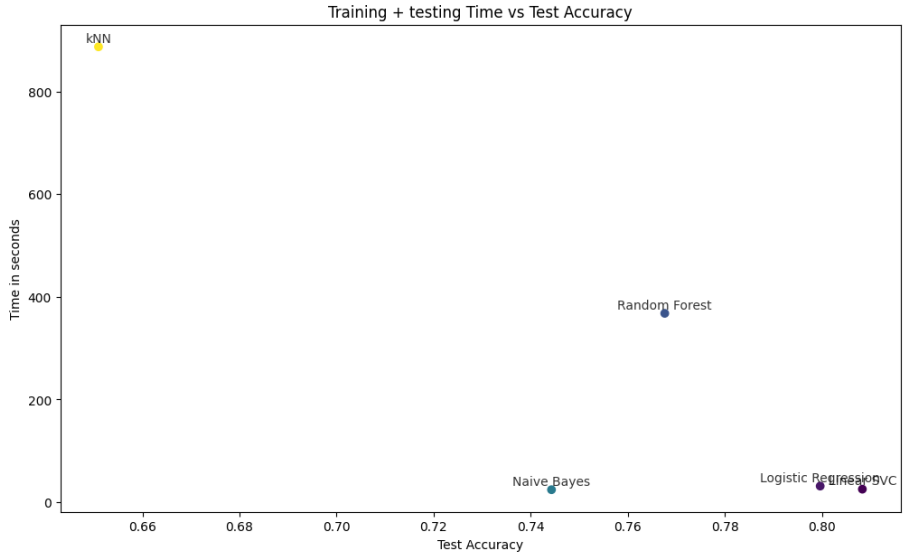

Figure 4: Time Accuracy trade-off

It is important to note that we made this plot trying to make it as fair as possible, by already changing some basic features of the different models to make them take into account that our dataset is balanced and made of a Sparse matrix. Results might vary depending on the choice of the parameters given to the different models.

In conclusion, we can see that Linear models consistently show good accuracy and their efficiency in terms of time makes them well-suited for optimization. On the other hand, Random Forest yields correct results and if we had more computer power, it would have been interesting to optimize it.

3.2 Description of predictive models

3.2.1 Multinomial logistic regression

The Multinomial logistic regression is a linear classification method that applies logistic regression to multiclass issues. In this method, the dependent variable should be categorical and the independent variables can be binary or continuous variables. This approach particularly works well when there is no hierarchy among the categories.

The goal of multinomial logistic regression is to model and predict the probabilities of multiple exclusive categories for a given set of input features. Thus, it is a discriminative model. Discriminative models are concerned with modeling the decision boundary between various classes or categories in the input data. They learn the mapping directly from the input features to the output classes without necessarily modeling the underlying probability distribution of the data. So they immediately estimate the posterior class probabilities :

$$\widehat{PR} = P(Y = j \mid X = x)$$

Let's go deeper into multinomial logistic regression. To link the probability of a class to $x^T\beta$, where $x^T\beta$ is $\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$, multinomial logistic regression employs a softmax function. The softmax function for class j is given by:

$$P(Y = j \mid X = x) = \frac{\text{Exponential of Raw Probability for class } j}{\text{Sum of s of Raw Probabilities for all classes}}$$

$$= \frac{e^{x^T\beta^{(j)}}}{\sum_{l=1}^{q} e^{x^T\beta^{(l)}}}$$

- $Y = j$: The event that the outcome variable Y takes the value j.
- $P(Y = j)$: The probability of observation i belonging to category q.
- $j = 1, \ldots, q-1$: The index representing classes from 1 to $q-1$.
- q: The total number of classes.

The denominator can be seen as the sum of the exponential values. If we want to isolate the contribution of category j in the sum, we can express the denominator as:

$$\sum_{l=1}^{q} e^{x^T\beta^{(l)}} = \sum_{l=1}^{q-1} e^{x^T\beta^{(l)}} + e^{x^T\beta^{(q)}}$$

Then the function becomes the one we saw in class:

$$P(Y = j \mid X = x) = \frac{e^{x^T\beta^{(j)}}}{\sum_{l=1}^{q-1} e^{x^T\beta^{(l)}} + e^{x^T\beta^{(q)}}}$$

After simplification, we have:

$$P(Y = j \mid X = x) = \frac{e^{x^T\beta^{(j)}}}{1 + \sum_{l=1}^{q-1} e^{x^T\beta^{(l)}}}$$

$$P(Y = q \mid X = x) = \frac{1}{1 + \sum_{l=1}^{q-1} e^{x^T\beta^{(l)}}}$$

- j= 1,..,q-1.
- $\beta^{(1)}, \ldots, \beta^{(q-1)} \in \mathbb{R}^p$: Parameter vectors to be estimated.

This ensures that the sum of the probabilities across all classes equals 1 and ensures that the raw scores are converted into probabilities.

Finally, to estimate the parameters of the model, we can use the maximum likelihood estimation. Be careful, contrary to logistic regression, the distribution of Y/X is multinomial and not bernouilli.The likelihood is:

$$l(\beta) = \sum_{i=1}^{n} \log\left(P(Y = y_i \mid X = x_i)\right)$$

Maximizing this sum is the objective in the maximum likelihood estimation process for multinomial logistic regression. The goal is to find the set of model parameters that maximizes the likelihood of observing the given outcomes across all data points. This is typically achieved using optimization algorithms.

To do this maximum likelihood, we have to have a loss function. In our code, as we use the parameters multi_class = "multinomial" and solver = "sag" (cf. section 4.1.6) the loss function is the cross-entropy loss for multinomial logistic regression:

$$L(w) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Where:

- $L(w)$ is the cross-entropy loss for logistic regression.
- N is the number of samples.
- y_i is the true class label for the i-th sample (0 or 1 in binary classification).
- $\hat{y}_i$ is the predicted probability that the i-th sample belongs to class 1.

3.2.2 Support Vectors Classifier

The Support Vector Classifier (SVC) is a generalization of the Maximum Margin Classifier (MMC). SVC is a powerful tool, used at first for classification tasks. SVC is particularly beneficial for text classification in Natural Language Processing (NLP) because it is efficient in high-dimensional data. Text data, once vectorized into numerical form like TF-IDF vectors, often consist of thousands of dimensions, each representing a term or word frequency.

SVC is known for its approach to classification through what is known as a soft margin. Unlike a hard margin classifier (MMC) that strictly divides classes without allowing any crossover, SVC introduces flexibility. It allows some observations to fall within the margin or even on the incorrect side of the dividing hyperplane. When classes are not linearly separable, the "budget" of permitted errors is purposefully designed to increase the classifier's robustness and generalization skills.

This soft margin approach is managed by a hyper-parameter, commonly denoted as C (C = 1 / "budget"), which controls the trade-off between having a large margin and minimizing the classification error. A smaller value of C allows more miss-classifications, promoting greater model generalization, while a larger C value encourages a stricter separation of classes.

The LinearSVC in scikit-learn uses a different formulation from the one in the lectures. Instead of solving the problem using the dual formulation with Lagrange multipliers, LinearSVC minimizes the primal form of the SVM problem. The optimization is given by:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n} \xi_i \tag{1}$$

$$y_i(\mathbf{w} \cdot x_i + b) \geq 1 - \xi_i \tag{2}$$

$$\xi_i \geq 0 \tag{3}$$

Here:

- $||\mathbf{w}||^2$ represents the square of the norm of the weight vector, which is related to the width of the margin. Minimizing this term effectively maximizes the margin between the classes.
- C is the regularization parameter that determines the trade-off between maximizing the margin and minimizing the classification error. It is the same C parameter you specify in `LinearSVC(C = 0.5)`.
- ξ_i are the slack variables representing the misclassification error for each data point. They are similar to the ϵ_i in the formulation provided in class but are not directly controlled; instead, they are managed implicitly through the C parameter.

The loss function used by the `LinearSVC` in scikit-learn by default is the 'squared hinge' loss. This is a variant of the standard hinge loss function that is used in the classical SVM formulation. The 'squared hinge' loss is defined mathematically for a given data point (x_i, y_i), model weights $\mathbf{w}$, and bias b as:

$$\max(0, 1 - y_i(\mathbf{w} \cdot x_i + b))^2 \tag{4}$$

Here's what each component means:

- y_i is the true label for the data point x_i, which is assumed to be either +1 or -1.
- $\mathbf{w} \cdot x_i + b$ is the raw model output before applying the sign function to make a prediction.
- $\max(0, 1 - y_i(\mathbf{w} \cdot x_i + b))$ is the hinge loss, which measures the distance of x_i from the margin when it's on the wrong side. If x_i is on the correct side of the margin, the hinge loss is 0.
- Squaring the hinge loss (hence 'squared hinge') penalizes the errors more heavily, contributing quadratically rather than linearly to the total loss. This makes the 'squared hinge' loss more sensitive to outliers and can lead to a model with a narrower margin, potentially increasing its sensitivity to noise in the data.

4 Improvement of the model

4.1 Choice of models features

This section is devoted to the different parameters we used to optimize our models. We will first speak about the most important parameters that we find in almost all our models to avoid redundancies. Then, we will look at the specific parameters of each method.

4.1.1 Amelioration of CountVectorizer

Inside the CountVectorizer, we can mention different parameters. A parameter that would be interesting to take into account is ngram_range. ngram_range is used to control the range of n-grams to extract from the text data. By specifying this parameter, we can choose if we want to treat each word in isolation or if we want to consider consecutive words. To get a clearer idea, let's take two examples. The first is ngram_range(1,1). This will simply consider unigrams, so simple words. Thus it will ignore the sequence of consecutive words and treat each word in isolation. Another alternative is ngram_range(1,2). In this case, it will consider unigrams and bigrams. This is really interesting since it provides more context and potentially captures phrases or expressions.

I study machine learning

Unigrams: "I", "study", "machine", "learning"
Bigrams: "I study", "study machine", "machine learning"

In our case we decided to use ngram_range(1,2) because it can be interesting to capture not only individual definitions (unigrams) but contextual relationships and co-occurrences of adjacent terms (bigrams).
Some tokens ("i", "am", "the", "a", "an" etc) may appear many times although they are of no use to estimate the content of a post. These words are called "stop words".

Numerous open-source stop word lists are available, and we initially initiated our project by employing the one recommended in the course through the NLTK package, implemented in our text_preprocessingfunction. However we found out that adding the English-language stop-word list of Sklearn increases our result significantly and thus decided to use it.

We were aware that those lists are generic and thus won't be optimal for our particular dataset. In section 4.2, we will describe our approach to optimize this list to better suit our dataset.

4.1.2 TF-IDF

Tf-IDF, known as term frequency-inverse document frequency, is a statistical metric that assesses a word's relevance to a document within a set of texts. In short, it analyzes the frequency with which a word occurs in a document and its inverse document frequency over a collection of documents. Therefore, simple words that are regular in many texts (if, what,...) rank low because they have little significance to that specific document, despite the fact that they may appear frequently. On the contrary, a word that appears many times in a text but not in others will have more importance because it seems relevant.

This can be done in Python with TfidfTransformer or TfidfVectorizer from sklearn.feature_extraction.text. So what is the difference between TfidfTransformer and TfidfVectorizer ? not much. Tf-IDF is always used after CountVectorizer. Thus, one can use TfidfTransformer after CountVectorizer or simply use TfidfVectorizer (it will do the two transformation at once).

There are multiple parameters that we can configure inside TfidfTransformer. One interesting parameter is 'norm', which regulates the term vectors' normalization. Its default value is "l2". This means it normalizes the term vectors using the "l2" norm (Euclidean norm). Another solution would be to use "l1". The choice must be based on the specific characteristics of our data and the problem we are trying to solve. For our linear SVC model and our multinomial logistic model, we have included this parameter in the grid-search and found that "l2" works better.

The other parameter we used is "sublinear_tf". "sublinear_tf" determines if a logarithmic scaling of the term frequency (TF) values is necessary prior to the TF-IDF transformation. It is useful if we want to reduce the effects of extremely high term frequencies. We think it should be good to incorporate it (sublinear_tf = True) in order to be sure to mitigate the influence of extremely common terms.

4.1.3 Lasso and Ridge penalty

In order to improve our multinomial logistic model, we also decided to add regularization techniques to prevent overfitting. There are plenty of regularization techniques such as subset selection, dimension, reduction, shrinkage etc. In our case, we decided to apply a shrinkage. This technique shrinks the estimated coefficients toward zero by introducing a penalty term during the training process which encourages a more parsimonious and generalizable representation of the underlying relationships between predictors and categories. We focused on techniques such as Ridge, Lasso and elastic-net penalty. These penalties are added to the loss function before doing the optimization.

$$\text{loss function} + \text{penalty}$$

There is a battle between these two components since it would be good for the penalty to have very small betas (close to 0 or 0) but this would not be a good fit for the loss function. Therefore, the penalty reduces the value of the coefficients. To choose the strength of the penalty, there is a tuning parameter lambda: λ. It should always be bigger or equal to zero and the higher it is, the more powerful the punishment.

In the case of Ridge regression the penalty shrinks some coefficients toward 0 (but cannot be 0) while Lasso can force some coefficient to be exactly 0 if λ is very large. This means that Lasso may perform variable selection. Let's take a look at both.

$$\text{The lasso penalty is: } \lambda \sum_{j=1}^{p} |\beta_j|$$

$$\text{The Ridge penalty is: } \lambda \sum_{j=1}^{p} \beta_j^2$$

The difference between these two penalties is the norm they use. Lasso uses the L1-norm and Ridge uses the L2-norm. L1 is more robust since it may reduce considerably some coefficients or even delete some of them. This property not only aids in feature selection but also results in simpler and more interpretable models.In contrast, L2 imposes a softer constraint on coefficients and may have more stability and handle correlated features more gracefully.

If we want to combine the strength of both Lasso and Ridge regression, we can use elastic-net. Elastic-net may be useful in the case of a large number of features, since it can manage situations where many features are potentially relevant, but only a subset should be selected for the model.

$$\text{The elastic-net penalty is: } \lambda \sum_{j=1}^{p} ((1-\alpha)\beta_j^2 \ + \alpha|\beta_j|)$$

α is a tuning parameter that chooses which punishment to focus on and should be between 0 and 1. There are two extreme cases. When $\alpha = 0$ the elastic-net becomes the Ridge penalty and when $\alpha = 1$ the elastic-net becomes the Lasso penalty. It is a useful tool to find a suitable compromise between variable selection and stability.

In Python, we can choose which penalty we want to use with our Multinomial logistic regression by specifying "penalty = 'l1'" , "penalty = 'l2'", "penalty = 'elasticnet'" or even "penalty = 'none'" if we don't want a penalty. By default, it is "l2" so the Ridge penalty.

To make this penalty available, we have to choose the value of the hyperparameter C. C is the inverse of the regularization strength and is used to control the trade-off between fitting the training data well and preventing overfitting. Therefore, a smaller value of C (or larger lambda) increases the regularization strength, encouraging sparsity in Lasso and preventing excessive coefficients in Ridge. A high C reduces the strength of the regularization. Thereby, C controls the trade-off between model complexity and overfitting. In Python, we simply have to specify a value for C between 0 and ∞. By default, C will be 1. By looking at the default value, we realize that if we don't mention both parameters there will be a regularization and it will be strong.

To be honest, we first tried different C by hand to see how the accuracy reacted. We tried some values between 1 and 10 (1,5,7,10) and played around the best value (7) and found C = 7.5 as the best one in term of test accuracy. Then, we submitted our logistic regression with C = 7.5 on kaggle and had nice result. To be sure of our result, we anyway did a grid-search and find 19 as the best C. We then applied the one-standard-error rule an found a C of 10.

4.1.4 Max iter and tol

Max_iter and tol are parameters related to the optimization algorithm used for finding the coefficients that minimize the loss function.

Max_iter is the maximum number of iterations the optimization algorithm is allowed to perform. The goal is that the algorithm has converged before the max_iter iterations. Thus, you have to choose a fairly large number but not too many either since it may induce overfitting. Indeed, if you set max_iter to an unnecessarily high value, the model may not be as able to generalize to new, unknown data because it will still be learning the noise in the training set.

Tol is the tolerance for the stopping criteria. It stands for the increase in log-likelihood that the algorithm needs to maintain in order to iterate further. If the change in the log-likelihood between two consecutive iterations is less than tol, the algorithm assumes convergence and stops. Therefore, a small tolerance increases the sensitivity of the optimisation algorithm to small changes, which may result in more precise solutions at the expense of longer computation times.

Max_iter and tol are working together to control the convergence of the optimization. When combined, they try to manage the trade-off between obtaining a sufficiently accurate solution and computational efficiency in logistic regression and related optimisation issues.

We decided to try different combinations of Max_iter and tol in our grid-search for the linear SVC and the best combination was the default combination: Max_iter = 1'000 and tol = 0.0001. For our multinomial logistic regression, to not have a too large grid-search that takes too much time we decided to fix the Max_iter parameter at 100'000 and search for the tol. We ended up a tol of 1^{-6}.

4.1.5 Class_weight

Finally, class_weight allows you to specify different weights to classes in the training dataset. The default value is 'None' which means all classes are treated equally. You can manually provide the weights for each class if necessary. An other interesting option is 'balanced'. 'balanced' uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as :

$$\frac{\text{number of samples}}{\text{number of classes} \times \text{np.bincount}(y)}$$

where:

- np.bincount(y) counts the occurrences of each element in the array y.

The idea is to assign higher weights to classes that are less frequent in the dataset, and lower weights to more frequent classes, thereby balancing the contribution of each class during the model training.

We first thought it was not necessary to use this parameter as our dataset seems already balanced (cf. section 2.1.1) but to be sure we put 'balanced' for both models (multinomial logistic regression and linear SVC).

4.1.6 Special parameters of multinomial logistic function

In our multinomial logistic regression, we also add parameters "solver" and "multi_class". First, as we are dealing with a multinomial logistic regression, we have to use a multinomial loss. It's the loss function employed to train a model in a way that it makes accurate predictions across multiple classes. The solvers in scikit-learn that are able to handle this are 'newton-cg', 'sag', 'saga' and 'lbfgs' . Then we can look at the size of our dataset. 'sag' and 'saga' are particularly useful when we are dealing with a large data set since they use the concept of stochastic gradient descent. Indeed 'sag' is for stochastic average gradient descent which is a variant of the stochastic gradient descent that supports multinomial loss. 'saga' introduces shuffling to 'sag', which can be beneficial in scenarios where the objective function is non-smooth.

We decided to use 'sag' because it is faster than 'saga' in terms of computational efficiency (a multinomial logistic regression with sag takes approximately 2 minutes while it takes 12 minutes with saga) and they have approximately the same results.

In conjunction with this option, you can use the multi_class parameter. It is used for specifying which strategy we want to use for our multiclass classification problems. There are 3 possibilities: 'ovr', 'multinomial' and 'auto'. 'auto' could be very good since it chooses between 'ovr' and 'multinomial' based on the nature of the problem and the solver.

4.1.7 Special parameters of linear SVC

In order to make some regulations in our model, we used some penalizations to prevent overfitting by penalizing the model complexity. In LinearSVC, this is controlled through the "penalty" parameter, which specifies the norm used in the penalization process. In SVC there are two types of norms :

- **l2**: This is the default. The L2 norm of the weights $||\mathbf{w}||^2$ is added to the loss function, which is defined as the sum of the squares of the weights. The regularization term in the objective function is given by:

$$\frac{1}{2}||\mathbf{w}||^2 = \frac{1}{2}\sum_{j=1}^{m} w_j^2$$

- **l1**: The L1 norm of the weights $||\mathbf{w}||$ is the sum of the absolute values of the weights. The regularization term in this case is:

$$||\mathbf{w}|| = \sum_{j=1}^{m} |w_j|$$

For the L2 penalty, the complete objective function for `LinearSVC` that it minimizes is:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w} \cdot x_i + b))^2$$

Here, C is a hyperparameter that controls the trade-off between increasing the margin size and ensuring that x_i lies on the correct side of the margin. A smaller value of C encourages a larger margin (more regularization), potentially allowing more misclassifications, while a larger value of C puts more emphasis on correctly classifying all training examples, which can lead to a model with a smaller margin (less regularization).

In our model, we decided to fix the penalty to "l2" since the "l2" penalty is the standard used in SVC. For the value of C, we decided to do a grid-search and found C = 1 as the best. In our linear SVC, we also add parameters "loss" and "multi_class". Multi_class is the same parameter as in the multinomial logistic regression but here there is not the "auto" possibility and the "multinomial" is replaced by the "crammer_singer". In the scikit-learn documentation, we found that crammer_singer is interesting from a theoretical point of view as it is consistent but it is rarely employed in practice as it rarely improves accuracy and is more costly to compute. To be sure, we decided to include this parameter in our grid-search and we found that the best model used "ovr".

For the loss function, we also decided to include it in the grid-search in order to know if it is better to use the hinge_squared loss or the hinge loss. The grid-search showed that the hinge_squared performs better in our case.

4.2 Enhancing Dataset Processing

4.2.1 Enhancing text formatting

As we didn't know much about text formatting, we used the text formatting proposed in the Guidelines Notebook as a starting point. This text formatting uses NLT (Natural Language Toolkit), which is a Python programming language package of tools and applications for statisti-

cal and symbolic natural language processing (NLP). However the text formatting function has some issues and can be enhanced.

The first major problem is that it uses a function of the package “BeautifulSoup”, that is made to transform the text by removing HTML elements. While it is useful when we retrieve text from a website as it removes html tags like “<h1>” next to each title, it poses a challenge for our dataset. HTML tags are closely related to some of the classes we try to predict (“css”,”html”, etc) and thus contain essential information to differentiate them from other classes that don’t contain HTML tags at all. In consequence, we decided to remove this line from the processing function, which led to an increase in accuracy.

Furthermore, we omitted the second line, "text.lower," given that our Dataset is made of a pre-cleaned version of the posts present on stackoverflow (cf. 4.2.1) and as such, the conversion to lowercase had already been applied during the initial cleaning process.

After adapting the function to make it fit our problem, we tried to improve our results by experimenting with the elements present in the replace_spaces and bad_symbols lists, as they are essential in the segmentation of the data. The first list is made of the elements that will be replaced by a space. It initially replaced the following elements: ’ / ’, ’(’, ’) ’, ’ ’ , ’ ’, ’ [’, ’] ’, ’ | ’, ’ @ ’, ’ , ’, ’ ; ’ by a space. After inspecting the text, we found that some words were adjacent to a backslash. For example, in the post with Id 127, starting with: “"how to store values in a hashmap/array,”, “hashmap/array” would be treated as a single token by the count vectorizer, rather than two separate tokens. Although the words "hashmap" and "array" might appear independently in other posts, the token “hashmap/array” probably only appears in this post. Consequently, we decided to remove the backslash by adding “\\” to the replace_space list, which led to an increase in our accuracy.

The second list is made of the elements that we want to remove from the text. The way it is written can lead to confusion as the ”^” next to the opening brackets means that this list will be made of all elements except those present in the list. Therefore, “0-9a-z” means we keep all the numbers and lowercase letters, “#+_” means that we keep those symbols. It is crucial to keep symbols like “#” and “+” in the context of our dataset, as they play a significant role in distinguishing between words such as "c#" or "c++". For example, to differentiate classes “c#” of the class “c++” , as the number of times “c#” appears in post labeled “c#” is higher than in the post labeled “c++” and vice-versa. Otherwise, only the token “c” would appear in both classes instead of “c++” or/and “c#”. We also tried to increase our accuracy by adding and removing several elements of the list and found out that keeping the points by adding [.] to it increased the results.

After all those changes, the last thing that we could modify to enhance our model performance is the stop_words list. Initially, we only used the one from the nltk package. However, after we found out that by specifying stop_words = “english” in the CountVectorizer function (cf 4.1.1) improved our score, we decided to combine both lists. We did it by selecting the unique el-

ements of each list, resulting in a personalized list to which elements could be added or removed. As discussed in section 4.1.1, stop_words lists are made by people with a deep understanding of English syntax and word corpus. However, considering the nature of our dataset, which includes "conventional" text but also contains coding expressions, we expected that by modifying it we might give some additional information to our models. Consequently, we went through the 259 elements of the combined list and removed those that were linked to coding expressions that we were already familiar with in the hope that keeping those words in the dataset could help the model perform better.

But sadly the removal of the following list of words from the stop-word list: (describe, call, else, except, for, find, full, from, get, none, or, re, system, while) didn't lead to an increase in accuracy.

4.3 Augmenting the dataset

While we explored strategies to augment our dataset in order to increase the performance of our models. We first thought about doing a Web Scraping algorithm on StackOverflow, to find labeled data that we could add to our original dataset. However, it would have been hard to implement such an algorithm and doing a lot of requests to scrap the thousands of Data posts raised concerns about potential restrictions or blocks from the Stack Overflow API.

Another option that we considered was to modify certain words with their synonyms in order to increase our dataset. It would have allowed us to generate meaningful posts with different words, thus diversifying our dataset while preserving the underlying meaning and structure. However, most translators wouldn't allow large corpus of text and the only alternative was to consider paid options for translating large corpus of text, but these were out of our budget.

After some additional research, we found out that in Google Sheets there is an option that allows us to change the language of a column even if it is a large corpus. Knowing that translating text to a different language and then to the original language usually replaces some words with synonyms, we decided to give it a try. We selected a random subset of 5 '000 rows (more rows will make it crash), equally distributed between classes and translated it to French and then back to English. Sadly, we realized that it just changed a few words in the data. We then decided to translate it from english to German, then to French and finally back to English. This time a lot of words were altered.

However, we found out a major problem inherent to our dataset, words related to coding and the labels tags present in the text like asp.net would be treated as misspelled words and translated into nonsense. We still tried to run a Logistic regression with our additional 5'000 posts and surprisingly, it increased our score from 83% to 87%. However, when we tried it into the final kaggle dataset, it decreased our score from 95% to 94%. Consequently, we decided not to incorporate this dataset into our final model as it didn't improve our model.

5 Results

In this section, we will present the results of our models with and without optimization. First, it is necessary to present the various tools employed for conducting the analysis of our findings.

Since we are doing a train-test split, we have the possibility to analyze the accuracy on the training set and on the test set. First, the accuracy on the training set allows one to know if the model has learned well from the training data. Often, it will produce a relatively high result and for a linear model it is of 1 or almost 1 which means our dataset is linearly separable. Then, the test accuracy allows one to assess if the model performs well on unseen data.

The goal is to have a model that not only performs well on the training set but also generalizes well to new, unseen data. It's important to exercise caution; if the model has a high accuracy on the training data but performs poorly on the test data, it may be due to some overfitting. It occurs when a model learns the training data too well, capturing random fluctuations in the data instead of the underlying patterns.

To select the hyper parameters that work best with our data and models, we used a Grid Search. It works by first randomly dividing the data into K folds of similar size. Then each K fold will be used as a test dataset, while a model is fit to the rest of the folds. This is done for each combination of parameters possible and a resulting MSE result is obtained.

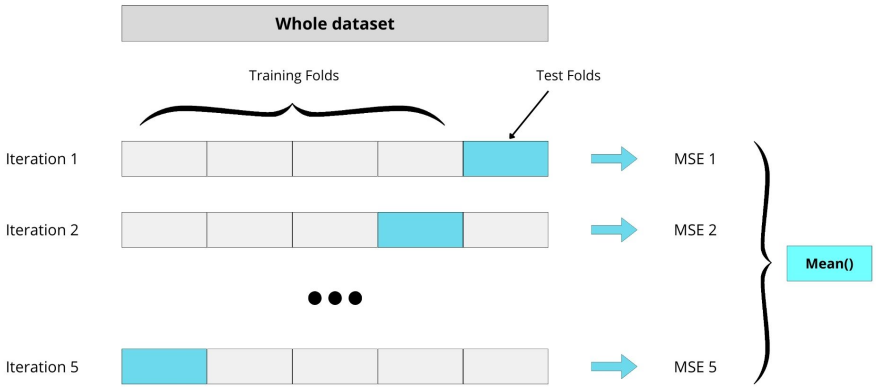

Figure 5: 5-Folds cross-validation for model's optimization

Regarding our parameters for the Grid Search, we used k = 5 as we have little computing power and recommended K values are usually between 5 and 10.

For the Multinomial Logistic Regression, we decided to conduct two distinct grid search to determine which penalty, either l1 or l2, performs better. The reason being that one of the solvers we wanted to test (sag) isn't compatible with the l1 penalty. In consequence, in the first Grid Search (1) we tested the parameters "norm" of the tf-idf, the strength of the penalty (parameter C), the tol parameter and the solver "sag" and "saga" with a fixed penalty (l2).

According to the Grid search, the best parameters are:

C	class_weight	max_iter	solver	tol	tfid_norm
19	'balanced'	1e5	sag	1e-06	l2

Table 2: Grid Search Results for multinomial logistic

According to the one standard error rule that helped us choose the most parsimonious model, the C value selected is 10:

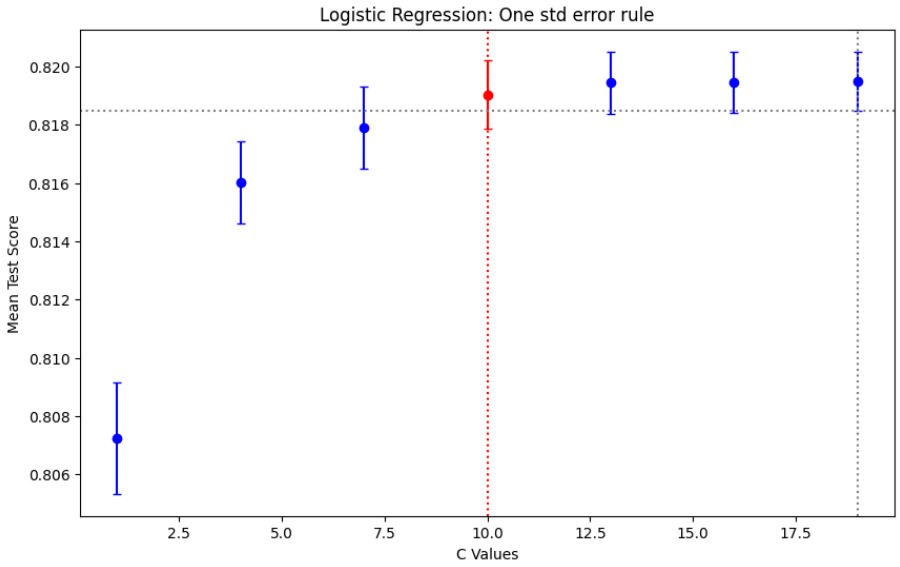

Figure 6: One-standard error rule for multinomial logistic regression (1)

In the second Grid Search , we wanted to try the "l1" penalty. Thus, we fixed "l1" with the "saga" solver and tried different values for the "tol" parameter and for the strength of the penalty (parameter C). Sadly, after 30 hours, for only 105 fits, the Grid Search was still running. The Saga solver being to slow, it is impossible for us to do a Grid Search on time so we abandoned the idea of applying the One standard error rule with solver Saga.

Regarding linear SVC we decided to fix the penalty to "l2" (cf. 4.1.3). The grid-search gave as best parameters:

C	max_iter	loss	multi_class	tol	tf-idf_norm
1	1000	'squared_hinge'	'ovr'	0.0001	'l2'

Table 3: Model Parameters

According to the one standard error rule the most parsimonious linear SVC model has a C value of 1. Unfortunately, it is the same C as the one selected without the one-standard-error rule. As shown in the graph, better models seem on the left (smaller than 1). Thus, we did another grid-search and one-standard-error rule but this time with smaller C. The result shows a C of 0.5 (cf. Annex, figure 10).

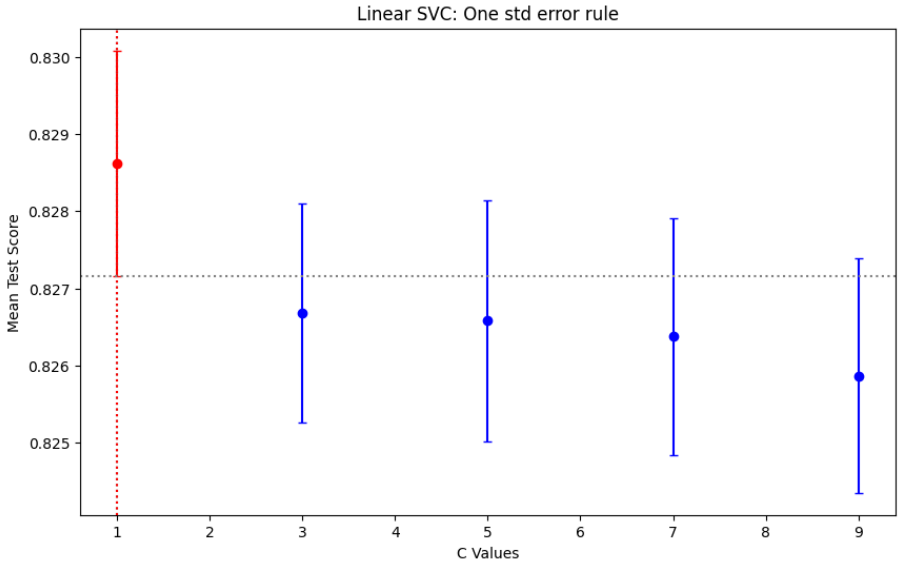

Figure 7: One-standard error rule for linear SVC

5.1 Table of results

We can see in the following table the influence of the improvements of the parameters and of the Text formatting on the final result. The optimized version (V2) slightly increases the results of the models. Even if the increase is around 1%, as explained before, top accuracy scores of the competition participants are around 95%, which means that little increase will be crucial for the final ranking. As we can see, for our best models (multinomial logistic and linear SVC) the Training accuracy and Test accuracy are very close which means that they are not overfitting.

Index	Model	Training Accuracy	Test Accuracy	Kaggle Accuracy
1	KNN	67.8%	65.1%	N/A
2	Naive Bayes	96.4%	74.4%	N/A
3	Random Forest	100%	76.8%	N/A
4	Logistic	99.8%	80%	N/A
5	Logistic Optimized	99.97%	81.62%	95.02%
6	Logistic grid-search	99.99%	81.62%	N/A
7	Linear SVC	100%	80.8%	N/A
8	Linear SVC grid-search	99.90%	82.05%	N/A

Table 4: Model Performance Comparison with V1 text formatting

Index	Model	Training Accuracy	Test Accuracy	Kaggle Accuracy
1	KNN	67.6%	65%	N/A
2	Naive Bayes	96.3%	74.4%	N/A
3	Random Forest	100%	77.2%	N/A
4	Logistic	99.7%	79.8%	N/A
5	Logistic Optimized	99.98%	82.5%	95.37%
6	Logistic grid-search	100%	82.6%	95.38%
7	Linear SVC	100%	80.9%	N/A
8	Linear SVC grid-search	99.85%	83.44%	95.13%

Table 5: Model Performance Comparison with V2 text formatting

If you want to check the code of these models, the index 1, 2, 3, 4, 7 are in the section 3.3 in our Jupyter notebook, the index 5 in section 4.5, index 6 in 4.6.1, and finally index 8 in 5.3.

Selection of the 2 Models for final Kaggle submission: Based on the scores of the different models, we opted for a safe option and a greedy option. The safe option is the Logistic grid-search (6), as it might perform better into new data as it is a parsimonious model. Our second choice is going to be our second best score on Kaggle The, Logistic Optimized (5).

5.2 Confusion matrices and Classification report

Complementary tools to make a diagnostic on the model are Confusion matrix and classification report. These good results are also visible using matrix confusion and classification report. Confusion matrix is a table that exposes all combinations of predicted outcomes juxtaposing them against the actual values. The Y axis is for the true values and the X axis is for the predicted value. Each cell within the matrix indicates the frequency of instances where the model predicted a particular class while the true class was something specific.

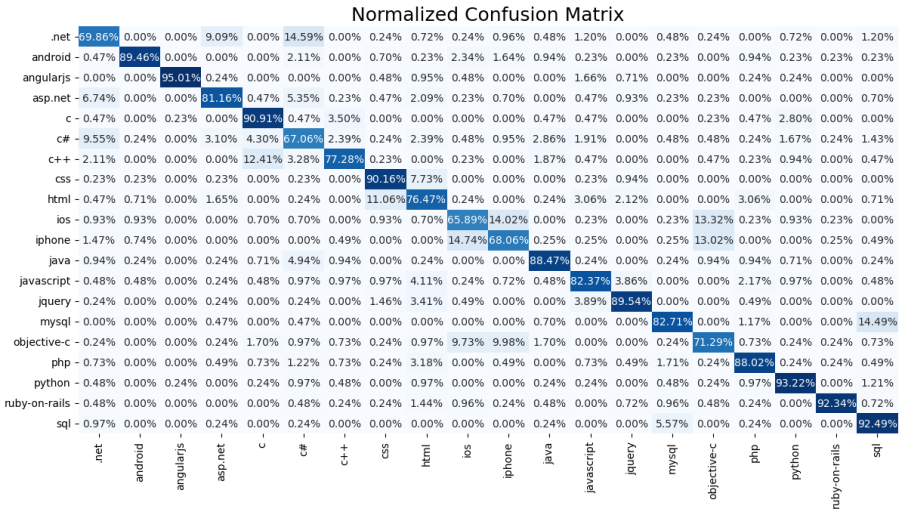

Figure 8: Confusion matrix of logistic grid-search

The goal is to have high numbers in the diagonal which will mean that we made good predictions. This structure allows you to analyze how well the model performs for each individual class and understand the types of errors it makes across the entire multi-class classification problem.

As we can see, our confusion matrix (figure 8) seems good. The biggest pourcentages are on the diagonal which means we estimate well the classes as in the other zones the percentages are close to 0. However there are still some classes that are confused. the biggest worries are between ios and iphone and between sql and mysql. We can also check the classification report. It gives you an overview of several metrics for every class, making it easier for you to evaluate the model's performance in numerous areas.

For each class, the report consists of precision, recall, f1-score and support. These metrics are calculated using the concept of true positive, true negative, false positive and false negative that make it possible to know if the predictions are right.

True positive: when a case was positive and predicted positive
True negative: when a case was negative and predicted negative
False positive: when a case was negative but predicted positive
False negative: when a case was positive but predicted negative

Classifier's exactness: Precision is the ratio of true positives to the total number of instances predicted as positive. It will give the percent of correct predictions.

$$\frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

Classifier's completeness: Recall is the ratio of true positive predictions to the total number

of actual positive instances. It is the ability of a classifier to correctly find all positive instances.

$$\frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

Average: F1-score is the harmonic mean of precision and recall, providing a balance between the two metrics.

$$2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Finally, support is just the number of occurrences of each class. Again, the results are very encouraging. Precision, recall and f1-score for each class are around 0.80 - 0.90.

6 Conclusion

In this report we presented a global approach to classify the tagging of user-generated articles on Stack Overflow. We tried to develop a highly accurate Machine Learning model in order to help Stack Overflow with the classification process, which is important to manage the high volume of data it receives regularly.

Our approach began with a detailed exploratory analysis of the data set, focusing on the distributions and patterns of postings and tags. We also tied to investigate the distribution of tags within the dataset, indicating a balanced foundation for our predictive modeling efforts and also explained the vectorization and tokenization process.

The selection of the models was a important step in our project. After testing lots of the models from our lecture, we focused on developing the Multinomial logistic regression and the Support Vector Classifier (SVC). Linear models, with their efficiency in terms of accuracy and time, proved to be well-suited for optimization. SVC, known for its effectiveness in high-dimensional data like text, was chosen for its flexibility and robustness in classification tasks.

Then we tried to improve these models by searching the bests parameters and improve feature engineering. We optimized the CountVectorizer and TF-IDF parameters, which were crucial for transforming textual input into a format acceptable for the Machine Learning models. We also used regularization techniques, especially Lasso and Ridge penalties, which were employed to prevent overfitting. Parameters like max_iter and tol were carefully adjusted to guarantee the optimization of the models to minimize the logistic loss function.

We achieved the best outcome with the Grid-Search Logistic Regression model, which delivered the highest test accuracy of 82.6%. However, In the public leaderboard of Kaggle the model, by its robustness and adaptability, gives us an exceptional Kaggle accuracy of 95.38%. Moreover, other models allowed us to get similar near results. Also the Linear SVC Grid-Search model which gives very satisfied results with a Kaggle accuracy of 95.13%, a bit lower than Logistic Regression model but still convenable.

In conclusion, our project has successfully utilized and optimized various machine learning models in order to try to predict tags for Stack Overflow posts, notably achieving significant accuracy with the Optimized Logistic Regression model. We also did a Random Forest model which showed respectable results, reaching 94.36% accuracy on Kaggle, but due to lack of time we could not develop it further. The next step for us could be to explore advanced approaches like Neural Networks, particularly inspired of Google's models, to potentially achieve even greater accuracy and efficiency. We would also have like to find a way to solve the confusion between [ios] and [iphone]. Overall it was a very interesting and stimulating competition, we learned a lot and had a lot fun.

7 Bibliography

References

[1] Mavroforakis, M. E., & Theodoridis, S. (2006). A geometric approach to support vector machine (SVM) classification. *IEEE Transactions on Neural Networks, 17(3), 671–682.* https://doi.org/10.1109/tnn.2006.873281

[2] Chrimni, W. (2022, 16 mai). *Support Vector Machines (SVM).* La revue IA. Available at: https://larevueia.fr/support-vector-machines-svm/

[3] Brackets and parentheses. (s.d.). *Overleaf, Online LaTeX Editor.* Available at: https://www.overleaf.com/learn/latex/Brackets_and_Parentheses

[4] scikit-learn. *Support vector machines.* Available at: https://scikit-learn.org/stable/modules/svm.html#svm-classification.

[5] Understanding TF-IDF: A simple introduction. (2019, 10 mai). *MonkeyLearn Blog.* Available at: https://monkeylearn.com/blog/what-is-tf-idf/

[6] Sklearn.linear_model.LogisticRegression. (s.d.). *scikit-learn.* Available at: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[7] Nothman, J., Qin, H., & Yurchak, R. (2018). Stop Word Lists in Free Open-source Software Packages. *Association for Computational Linguistics*, 7–12. https://doi.org/10.18653/v1/w18-2502

8 Annexe

Figure 9: Wordcloud for words distribution in classes

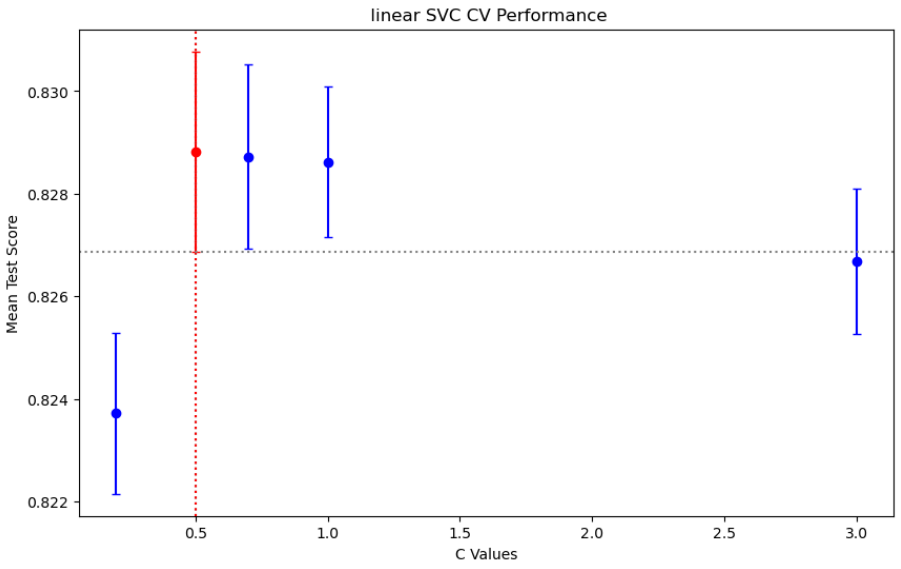

Figure 10: Second one-standard error rule for linear SVC