**Machine Intelligence - CO472**

**Automatic label prediction for GitHub issues**  Tejas R

16CO148

B.Tech 8th semester

9845762892

[tejas1908@gmail.com](mailto:tejas1908@gmail.com)

**Abstract**

GitHub is one of the most popular version control platforms in use today. It facilitates developers around the world to collaborate and contribute to open source repositories. One of the applaudable features of GitHub is GitHub issues, where contributors and clients can raise issues regarding any repository which needs the developer's attention. With thousands of such issues, it becomes difficult for contributors and owners to find issues requiring their expertise. In this project, we suggest a methodology to automatically predict labels for these issues. This helps in improving the developer productivity as the issues are now segregated and further reduces the time required for delivery of the product. The problem statement is formulated as a multi-label classification problem. We use various supervised machine learning models trained on the issues mined from the top 100 GitHub repositories. We perform an exploratory data analysis of the issues to filter out the most frequently used labels. We finally evaluate the performance of our predictions in terms of various metrics like the F1 score and the Jaccard score.

**Keywords**

Mining software repositories, natural language processing, Multi-label text classification, GitHub issues, classifier chain, multilayer perceptron, supervised learning, recommender system.

**1. Introduction**

Multi-label classification is a supervised machine learning problem where each instance of data is associated with one or more labels. This is a variation of the multi-class classification problem in which each instance is only associated with one label, which can take only one value among a predetermined set of classes. In the generalized multi-label classification the target value for each instance consists of a subset of labels. Multi-label classification is used for various applications like creating user profiles, sentiment analysis of user opinions, and image labeling.

In natural language processing multi-label classification is used for automatically tagging documents into a subset of previously established categories. In this project, we use multi-label classification for automatically assigning labels to GitHub [1] issues. Labels for GitHub issues are an integral part of the repository. Open source projects such as Google Kubernetes have thousands of contributors working on developing the product and, as a result, have thousands of issues in the repository. Consider a scenario in the absence of labels for those issues; it would be a tremendous task to find and resolve an issue relevant to your expertise among the entire issue pool. Labeling of issues makes it possible for the developers to filter issues based on labels and tending to the problems addressed quickly. This solves only one half of the problem as presently, GitHub requires the issues to be labeled manually. This might not always be the case, as only a part of the developer community follow good software development practices. As a result, a significant number of issues remain unlabeled and hence arises the need for automatic labeling of issues.

A recommender system, as such, can be used as an add-on in GitHub or as a Chrome extension, which could automatically assign labels to the issue as and when the user raises one. The user can further add more labels based on his domain-specific knowledge. This project aims at building a recommender system using supervised learning algorithms, that is capable of predicting labels for GitHub issues with decent accuracy. The entire project focuses on three critical sub-problems. Firstly, data acquisition, the data required for training, is mined from top GitHub repositories and compiled into a raw data source. The second problem is data cleaning and data pre-processing. We have addressed this problem and proposed a detailed workflow for pre-processing the input text data and the ground-truth label data. Finally, a comparative study of the performance of the various models is performed based on multiple performance metrics like F1 score, Jaccard score, and Hamming loss.

**2. Literature Review**

A similar problem of predicting tags for StackOverflow (SO) questions have been studied and researched upon already. Discriminative models [2] have been used for automatically predicting tags for Stackoverflow questions. They have used a variant SVMlight to train on the SO questions. This model is trained to predict 834 popular tags belonging to over 2000 questions. The test set is precisely balanced to have an equal number of positive and negative samples for each tag. The output of SVM is used, and the similarity between the question and the modeled questions is calculated, and the tags of the model having maximum similarity with the question are allotted to the question.

In [4], researchers working on the same problem discuss the pre-processing strategies used, one which cleaned the text in series and another, which just used pre-existing libraries. They then vectorized the text utilizing the Bag Of Words [5] vectorizer. Pre-processing the text was done by removing the punctuations, converting it to lowercase, removing the HTML tags and stopwords, and finally stemming. Feature selection is performed using a Linear support vector classifier, and an SVM is used for classification.

In [6], they have used a variation of co-occurrence between the tags and the text. The tags with the highest activation among the words are predicted. These activations depend upon the prior probability of tag occurring based on frequency and recency. [7] presents a comparative study of algorithms for the Stackoverflow tag prediction problem. They have created a custom lexicon of computer science-related keywords. They used the Nearest neighbor search (NNS), tag-keyword co-occurrence models, and fuzzy NNS and compared their performance on the SO data using the F1 score for evaluation. J. Nam et al. [8] have proposed Neural network architecture for multi-label classification. Pairwise ranking loss is used for training instead of cross-entropy, and their performances are compared. They also come up with a thresholding solution to obtain optimal F1 scores.

**3. Problem statement**

This project aims at building a recommender system that is capable of automatically predicting labels for GitHub issues, based on the content of those issues, with reasonable accuracy so as to be used as an add-on plugin with GitHub to aid and facilitate the repository owners and contributors. This tool could prove useful to developers as currently, GitHub requires the developers to manually label the issues which might not always be followed. This results in a considerable chunk of unlabeled issues which have to be scrutinized and analyzed to be assigned to the concerned domain. This takes up a substantial amount of time and effort, which in turn reduces the software productivity. A recommender system, as proposed, could solve these issues by fast-tracking the labeling process by automating it.

**4. Multi-label Classification**

Multi-label classification is a variant of multi-class classification and falls under the domain of supervised learning. Unlike multi-class classification in which each instance is associated with only one label which can take only one among n classes, in multi-label classification, each instance can be associated with a subset of the n classes. For example, consider the problem of classifying an animal into a cat or a rat. Given a dataset of images of only cats and rats, an image can only belong to one class i.e a cat or a rat. This problem belongs to the multi-class category. Now consider the problem of assigning movie genres to a movie. A movie can simultaneously belong to thriller and horror. This makes it a multi-label classification problem. Multi-label classification is an essential problem as it is a generalized version of the multi-class problem, which assumed the target classes to be mutually exclusive. Examples of use cases include predicting movie genres based on the plot or poster of the movie, recognition of user sentiment based on feedback, etc.

**4.1 Multi-label classification methods**

This can be approached in the following three ways:

1. Problem transformation methods: The given multi-label problem can be divided into multiple single label problems. This can be further categorized as the following two.

* Binary relevance, in which an ensemble of binary classifiers is trained for each class. Hence the total number of classifiers used in this model is equal to the number of unique labels in the training set. The union of all the predicted classes is considered as the output label for the multi-label problem. One disadvantage of this method is that it assumes the labels to be mutually exclusive and ignores possible dependencies among the labels.
* Label powerset, in which each element in the powerset of the labels is considered as a single class and a multi-class classifier is trained on those classes. The number of classifiers required is 2**|L|** where L is the set of distinct labels in the training data. This method considers the correlations between the labels but becomes computationally expensive for problems with large label sets.

1. Ensemble methods: In this approach, a chain of binary classifiers is trained over the training data. Consider a chain of m classifiers as follows: C0, C1,... Cm-2, Cm-1. Input to the classifier Ci consists of the training data plus the outputs of the previous classifiers C0 to Ci-1. This method takes into account the correlations between the labels and is also computationally reasonable.
2. Problem adaptation methods: This involves modifying the single-label classification algorithms by changing the decision functions such that it is suitable for classifying multiple labels. Multi-label K-nearest neighbors ML-KNN is one such algorithm.

In this project, we explore Binary relevance methods using classical classifiers, Classifier chains, and Multi-layer perceptron (MLP) for predicting the labels and perform a comparative study of the performance of each. We use the following evaluation metrics for comparing the models: Jaccard score, Hamming loss, Precision, and F1 score.

**5. Proposed Methodology**

The following section contains details of the proposed solution to the problem statement, implementation details, platform, and programming language used, dataset and evaluation metrics used, and finally, the results obtained and an analysis of those results.

**5.1 Conceptual model**

The entire project can be divided into three major phases. We focus on the following three sub-problems in detail and perform experimentation to obtain the most optimal results:

1. Data Acquisition: The data required for training is obtained using the GitHub API. Issues of the top 100 most popular repositories and collected and consolidated into a single file.
2. Exploratory data analysis (EDA) and data pre-processing: The collected raw data is then carefully analyzed, and data cleaning is performed to obtain clean, structured information that can be used for model training.
3. Comparative study of models: The cleaned data is then used for training, and a bunch of classification models and algorithms are used on the data. Their performance is then compared, and the model which delivers the best accuracy, is then suggested for use.

The overview of the entire methodology can be represented, as shown in figure 1.

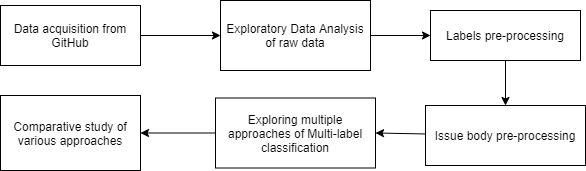


Figure 1: Overview of data flow in the proposed methodology

The trained recommender system is expected to work on unlabeled data, i.e., unlabeled GitHub issues, and automatically predict suitable labels based on the content of those issues. For an unseen instance, the recommender system follows the following workflow to predict the labels.

The labeled issues are pre-processed and used for training various multi-label learning models. The training process gives as output the best classifier, which is then used on the unlabeled data.

Unlabeled data follow the same steps of pre-processing as the labeled data. The unlabeled data is first cleaned, and punctuation is removed, followed by lemmatization and removal of stop words. The words are then embedded into feature vectors, and this is then used as input to the classification process. The best classifier output by the learning process is used to predict the labels for the unlabeled data. Figure 2 represents the entire process as a flowchart below.

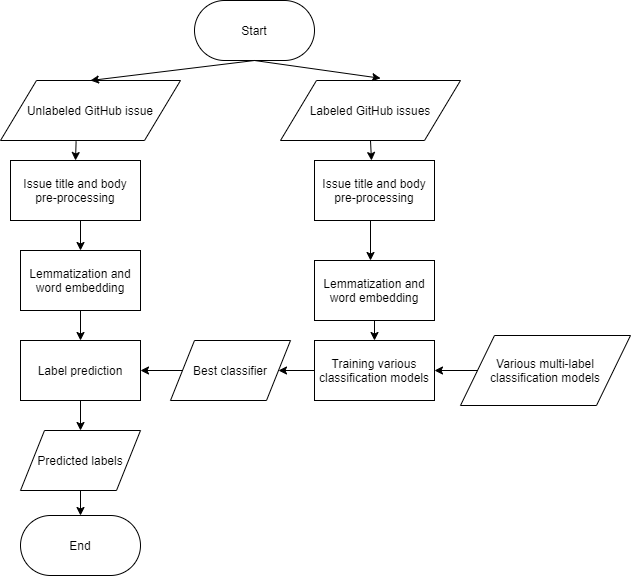


Figure 2: Flowchart of data flow in the inference process

**5.2 Algorithmic model**

This section presents a detailed explanation of the algorithms followed in the project. The raw GitHub issue data is first cleaned to structure it in a format that is usable for training the classification models. Raw data is unstructured and has a lot of missing and redundant values. The first step to data cleaning is removing the missing values and redundant values. The following is the algorithm developed to remove the irrelevant information.

**5.2.1 Drop irrelevant information - Algorithm**

1. From the raw issues, data input drop the irrelevant data columns.
2. Check whether the raw data has missing data in any of the cells.
3. If yes, remove all the rows with missing data values.
4. Otherwise, proceed to step 3.
5. Check whether the data has duplicate row entries and drop them accordingly.

Data resulting after this is devoid of missing data and duplicates. The next step in data pre-processing is the cleaning of labels. The labels in raw data are highly inconsistent. GitHub currently requires developers to manually attach labels to issues. This means that the type and format of labeling depend upon the developers. As GitHub doesn't restrict users from using their own custom labels, there are a variety of labels attached to issues. Besides this, there is a possibility of one or more forms for the same label. Consider, for example, a label "bug." This can be written as "bug" or "Bug" or "[Type] Bug", etc. Thus arises the need for cleaning the labels for issues. The following algorithm is followed to remove inconsistencies and clean the label data.

**5.2.2 Label pre-processing - Algorithm**

1. Convert all the labels to lowercase.
2. Remove unwanted characters like ‘<’, ‘]’, etc.
3. Check if the labels have numbers in them.
4. If numbers are present in the beginning, strip the numbers from the labels.
5. If the numbers are present in the end or amidst text, retain them.
6. Remove inconsistencies in the labels and reduce all the labels to one single form.
7. Get a list of all the unique labels among the total label set and compute the frequency distribution of each label.
8. Filter out the top 100 labels from the unique label set and store it in top\_100\_labels. The model will be trained to predict only these labels.
9. Iterate through all the issues in the dataset and remove any label which is not present in top\_100\_labels.
10. Remove any issue instance which doesn’t contain at least one label in top\_100\_labels.

After pre-processing the labels, we have a clean label set which consists of only the top 100 labels. The following labels are some of the most frequent labels among the top 100 labels:

* bug
* enhancement
* triaged
* help wanted
* module.

To filter out the top 100 labels from the complete set of labels, which is almost four hundred thousand, we performed a frequency distribution and filtered out the top 100 most frequently occurring labels. Figure 3 is a frequency distribution plot of the ten most frequently occurring labels.

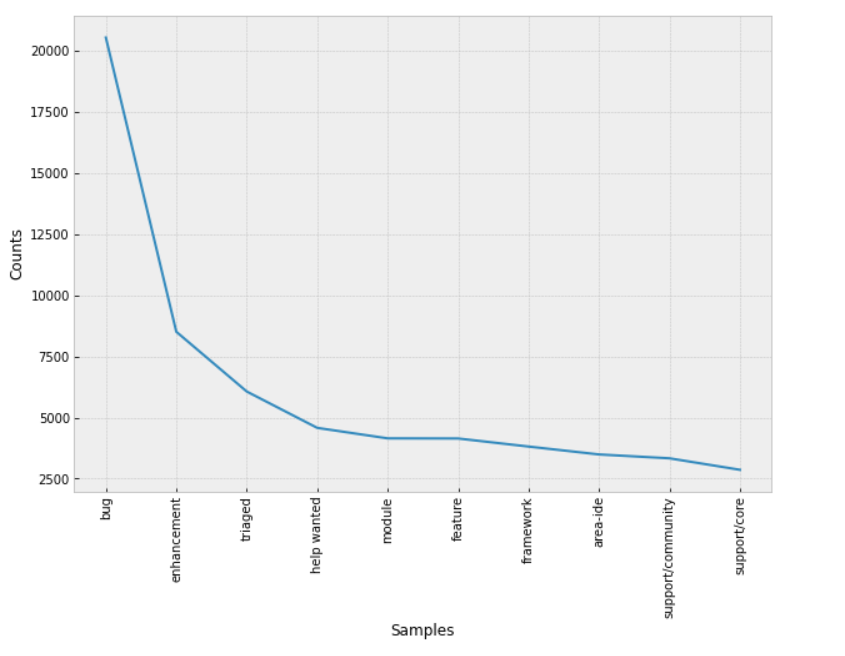


Figure 3: Frequency distribution plot for top 10 labels

This leaves us with the cleaning of issue data. Every issue in GitHub consists of a title and a body. We will use text from both these fields to predict the label for the issue. Hence we need to pre-process and clean both the title and the body. Raw data in the body field contains HTML tags, punctuations, short forms (what’s instead of what is), people tags (@name ), URLs to code snippets or errors, and escape sequences. The following algorithm is followed to remove all the above noise from the issue data.

**5.2.3 Issue data pre-processing - Algorithm**

Iterate through all the issues and perform the following steps for both the title and the body fields of the issues.

1. Remove HTML tags and comments (<!-- -->)
2. Remove elements that tag other developers in the issue. These elements are of the form of @developer\_name.
3. If, after the removal of HTML tags from the data, there are still URLs present, remove them.
4. Remove escape sequences like newline (\n) and carriage return (\r). These are irrelevant for prediction.
5. If short forms like “what’s” are present, replace them with their complete form.
6. Remove punctuation like #, !, $, etc. from the data.
7. If there are numbers in the data, remove them as they don’t contribute to the classification process.
8. After removal of numbers, if there exist words with a single letter, remove them.
9. Lemmatize the words in the issue data.
10. Remove any stop words present in the issue data. But if any word present in issue data is a part of any label, then don’t remove it.

Lemmatization is the algorithmic process of converting all the inflected forms of a word into its base form or the dictionary form. This base form is called the lemma of the word. Lemmatization correctly identifies the use of the word in its context and reduces it into its lemma based on its part of speech. Stop words are those words in any language which are found almost in all the documents and hence do not add any value to language modeling. These are the words that appear the most in any document and are therefore removed from the documents to improve computation speed and accuracy.

After lemmatization and removal of stop words, the issue data is now reduced to a sequence of lemmas. This cannot be understood by the classification model. Hence we converted the sequence of words into a sequence of numbers. We use term frequency-inverse document frequency (tf-idf) for embedding words. Term frequency refers to the frequency of occurrence of the word in the document under consideration, and inverse document frequency is the inverse of the rate of word occurrence in all the documents in the corpus. Inverse document frequency measures the uniqueness of the word in the corpus. Hence, tf-idf is a measure of the importance of the word to that document.

**tf-idf i,j = tf i,j log(N/ dfi,j) (1)**

tf i,j  = number of occurrences of the word i in document j

dfi,j = count of documents in which word i appears

N = total number of documents in the corpus

Equation 1 represents the tf-idf statistic for a word **i** in a document **j**. All the issue data is converted using tf-idf word embeddings. This leaves us with a sequence of floating-point numbers for each issue data. This can be used for training the classification models. We choose the top 1000 words having the highest tf-idf statistic from the issue title and issue body field each. Hence each row is now represented as a 2000 dimensional vector, where the title and body contribute 1000 features each. The labels of issues are now converted to a one-hot representation—the results in a target vector of length 100.

**5.3 Dataset**

The dataset required for training is obtained from GitHub repository mining. We made a list of the top 100 repositories in GitHub [3]. Using GitHub API and python libraries like requests, we mined issues of all these repositories and stored them into a CSV file. Raw data, which was extracted, has around 165 thousand issues. Figure 4 below is a sample of the raw issue data mined from GitHub.

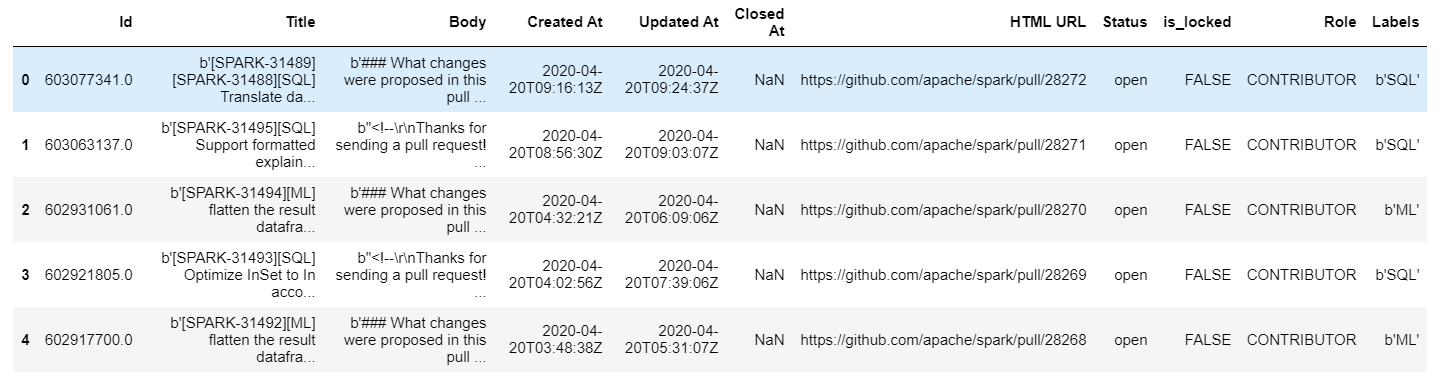


Figure 4: Sample of raw data

The following fields were recorded :

1. Issue ID
2. Title
3. Body
4. Issue creation date
5. Issue updated date
6. Issue closed date
7. URL to the issue
8. Status of the issue (open or closed)
9. Is\_locked (field to denote whether the issue or locked )
10. Role of the user who raised the issue
11. Issue labels

After pre-processing the data, dropping missing and duplicate entries, and filtering out only the top 100 labels, we are left with 95 thousand issues that can be used for training. We also have columns which we do not require for training. We drop the irrelevant columns and are left with clean and redundant data. Figure 5 represents a sample of data after cleaning and pre-processing.

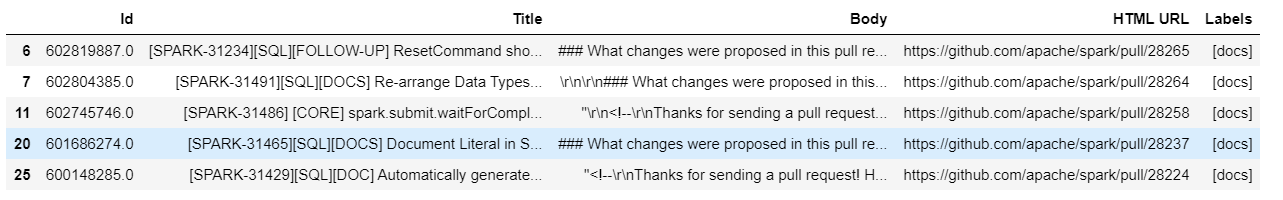


Figure 5: Sample dataset after cleaning and pre-processing

**5.4 Implementation**

The proposed solution is implemented using python on a Jupyter notebook. The following tools and libraries are used for data acquisition:

1. Python **requests** library
2. Python **csv** library
3. GitHub API

Apart from these, various other python libraries and packages were used for data exploration, visualization, data manipulation, and training the model.

1. Python **pandas**, **pickle** and **NumPy** for data manipulation
2. Python **nltk** and **re** for data pre-processing
3. Python **sklearn** for feature extraction, training, and evaluation.

The cleaned dataset is used for processing the labels. As seen above in figure 3, we filter out only the top 100 labels and retain them while removing the rest. Then we proceed to clean and pre-process the title and the body fields. The following is a sample body field before pre-processing:

*‘### What changes were proposed in this pull request?\r\nThis PR is the follow-up PR of* [*https://github.com/apache/spark/pull/28003\r\n\r\n-*](https://github.com/apache/spark/pull/28003/r/n/r/n-) *add a migration guide\r\n- add an end-to-end test case.\r\n\r\n### Why are the changes needed?\r\nThe original PR made the major behavior change in the user-facing RESET command. \r\n\r\n### Does this PR introduce any user-facing change?\r\nNo\r\n\r\n### How was this patch tested?\r\nAdded a new end-to-end test\r\n\r\n’*

After applying the proposed algorithm for title and body pre-processing, the following is the cleaned body field:

*‘change propose in pull request pr follow up pr add migration guide add an end to end test case change need original pr make major behavior change in user face reset command do pr introduce user face change no patch test add new end to end test’*

We see that all stop words, escape sequences, URLs, and numbers are removed from the text, and all the words are reduced to their lemmas. After title and body pre-processing, the text is vectorized and represented by their tf-idf values.

We then perform topic modeling using the Latent Dirichlet Allocation (LDA) algorithm to cluster words and assign them to various topics based on their tf-idf values. Figure 6 shows a few topics observed among the issue body text. This gives us a taste of how the words are related to each other, their correlation, and inter-dependency, which further helps us in label prediction.

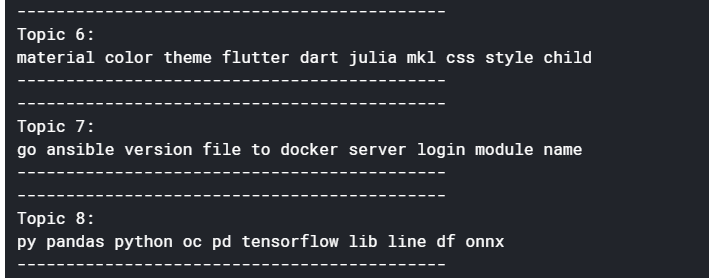


Figure 6: Sample topics obtained using LDA topic modeling algorithm

**6. Experimentation and Results**

After processing the title, body, and labels, we converted them into numeric form. The title and body fields are each represented by a 1000-dimensional vector and stacked together. Hence the input feature vector has a dimension of 2000. The target variable labels are represented as a 100-dimensional vector in the one-hot encoded format. We started off the training process by experimenting with classical classifiers. Their performance was compared using various evaluation metrics. All the conventional models are from **python’s sklearn** library. Table 1 below is a comparison between the performance of the multiple models trained.

The formulae for the Jaccard index and Hamming loss are as follows:

**J(A,B) = | AB| / | AB|**  **(2)**

Where A= Set of ground-truth labels

B = Set of predicted labels

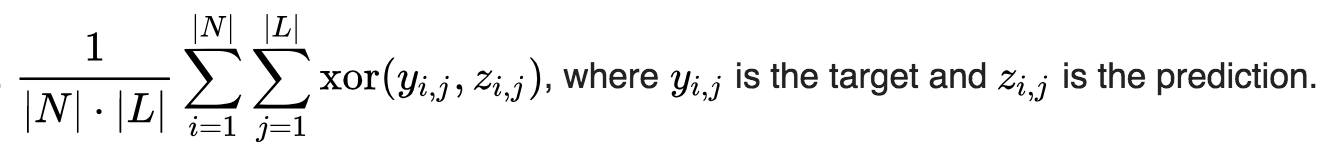
 **(3)**

Figure 7: Equation for Hamming loss

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Classifier \ Metric** | **Precision (%)** | **F1 score (%)** | **Jaccard Index(%)** | **Hamming loss** |
| **Logistic regression** | 48.26 | 44.17 | 39.96 | 1.19 |
| **Multinomial Naive Bayes** | 36.29 | 37.43 | 31.90 | 1.71 |
| **Linear SVM** | **53.24** | **49.66** | **44.90** | 1.18 |
| **Perceptron** | 48.00 | 47.65 | 41.04 | 1.67 |
| **Passive Aggressive classifier** | 48.78 | 47.67 | 41.61 | 1.52 |

Table 1: Performance comparison of various classical classifiers

We see that a linear Support Vector Machine (SVM) gives the best output among the classical classifiers. We are only studying Precision and F1 score and not recall because it is more important for the predicted labels to be correct than to predict all the labels in the ground-truth. As this system is only a recommender system to automatically predict labels, we only require it to suggest suitable labels, if not all the labels exactly in the ground-truth.

We also experimented with an MLP. The neural network has 2000 neurons in the input layer and 100 neurons in the output layer. The number of hidden layers and the number of hidden neurons are tuned and experimented. The output of the MLP is a 100-dimensional vector of floating-point integers. These integers are the probabilities of the input instance belonging to the various label classes. By default, the sigmoid activation function sets the output to 1 if the likelihood is greater than 0.5 and sets it to 0 if the likelihood is lesser than 0.5. This is not useful for our use case, as the probabilities are all much smaller than 0.5. Hence we iterate through values from 0.01 till 0.9 in steps of 0.01, and using this as the threshold, we set the output of the neurons are 1 or 0. We calculate the F1 scores for all these threshold values and choose the threshold, which gives the maximum F1 score. By varying the model architecture and the input features, we obtained the following results from the MLP model. Table 2 contains the details of the MLP model:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Hidden layer neurons** | **Threshold value** | **Precision (%)** | **F1 score (%)** | **Jaccard Index(%)** | **Hamming loss** |
| **200** | **0.14** | 54.36 | 57.55 | 49.82 | 1.55 |
| **400** | **0.12** | 54.39 | 58.37 | 50.30 | 1.57 |
| **600** | **0.1** | 54.67 | 58.64 | 50.55 | 1.57 |
| **800** | **0.11** | 55.45 | 58.79 | 51.00 | 1.51 |
| **1200** | **0.1** | 55.82 | 59.03 | 51.34 | 1.48 |
| **2000** | **0.11** | 57.00 | 59.77 | 52.24 | 1.42 |
| **4000** | **0.08** | 56.67 | **60.03** | **52.30** | 1.45 |

Table 2: Performance of classification by an MLP

We see that by fine-tuning the network architecture and the threshold value, we are able to improve the prediction accuracy of the MLP model in terms of the F1 score and Jaccard index. This model can further be fine-tuned for better results.

We now explore the performance of the Classifier chain models. In these models, we use a chain of classifiers for classification of labels. Each classifier in the chain uses the training data as well as the classification output of the previous classifiers in the chain as input for classification. As Linear SVM performed best among all the classical classifiers, refer to Table 1, we construct a classifier chain of Linear SVMs for multi-label classification. Even though SVMs output the label vector as 0s and 1s, the output of the ensemble model, which is the mean of all the outputs will be floating-point numbers. Hence arises the need to study the performance of the model based on the threshold value. As in MLP above, we change the threshold value and observe the F1 score and choose the threshold that gives the maximum F1 score. The following Table 3 consists of the results obtained by using a classifier chain of Linear SVMs. We experiment with the chain length and the threshold value and study the evaluation scores.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Chain length** | **Threshold value** | **Precision (%)** | **F1 score (%)** | **Jaccard Index(%)** | **Hamming loss** |
| **5** | **0.1** | 56.32 | 59.00 | 51.20 | 1.76 |
| **10** | **0.15** | 58.82 | 60.30 | 53.06 | 1.50 |
| **15** | **0.15** | **59.02** | **60.60** | **54.23** | **1.47** |

Table 3: Performance of classification of a Linear SVM classifier chain

We observe that classical classifiers that did not perform satisfactorily are now performing to mark of neural networks when used as a classifier chain. These ensemble methods can further be explored and compared with neural networks. We have come up with a methodology to automatically recommend labels to GitHub issues. We have also examined the performances of various models in this task. All these models have to be studied further, and a comparison of their training and inference times has to be performed before choosing the best model for the task.

**7. Conclusion**

In this project, we have explored and focused on three sub-problems. Firstly, data acquisition from GitHub API and storable in a CSV format. Secondly, we proposed algorithms to clean and pre-process the various fields of GitHub issue data. All the unwanted information was removed. We took care of anomalies and corner cases and converted the title, body, and label fields into a clean, usable form. Finally, we turned the issue data into a numeric form by using the tf-idf vectorization and trained various multi-label classification models on this data. We then explored and studied the performance of various models and compared them to one another.

We have shown in detail the entire procedure, from start to finish, to build a recommender system to automatically predict labels for GitHub issues. This can save a lot of time as the developers don't have to invest their time manually labeling the issues. Lots of future research work can be done in this area. Firstly we can employ deep learning models and study their performance on this data. State of the art algorithms like BERT and its family of algorithms can also be explored on this issue data. A further comparative study has to be performed among the classification models based on the training and inference time taken. After obtaining the best model with acceptable accuracy and reasonable inference time, we can employ it as a GitHub plugin for automatically labeling issues.

**References**

[1] T. Preston-Werner, “GitHub,” https://github.com/, 2020, [Online; accessed 12-April-2020].

[2] Saha, Avigit K., Ripon K. Saha, and Kevin A. Schneider. "A discriminative model approach for suggesting tags automatically for stack overflow questions." Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013.

[3] D. Smooke, “Hackernoon,” https://hackernoon.com/githubs-top-100-most-valuable -repositories-out-of-96-million-bb48caa9eb0b, 2020, [Online; accessed 16-April-2020].

[4] J. R. C. Gonzalez, J. J. F. Romero, M. G. Guerrero, and F. Calderon. “Multi-class multi-tag classifier system for StackOverflow questions.” 2015 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC).

[5] Z. S. Harris, “Distributional structure.” Word, 1954.

[6] C. Stanley and M. D. Byrne, “Predicting tags for StackOverflow posts,”

in Proceedings of ICCM, 2013.

[7] J. Hong and M. Fang, “Keyword extraction and semantic tag prediction,” 2013.

[8] J. Nam, J. Kim, E. L. Menc'ıa, I. Gurevych, and J. F¨urnkranz. "Large-Scale Multi-label Text Classification - Revisiting Neural Networks." Springer-Verlag Berlin Heidelberg 2014.

[9] S. Schuster, W. Zhu, and Y. Cheng, “Predicting tags for StackOverflow questions,” 2013.

[10] Morrison, Patrick, and Emerson Murphy-Hill. "Is programming knowledge related to age? an exploration of stack overflow." Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on. IEEE, 2013.