FIT5149 S2 2020 Assessment 2: Scientific Document Classification

REPORT

Group no. 57

Team:

MD. Saadman Hossain

S M Irfanul Mazid

Table of Contents

Introduction:

Natural language processing can be used as a machine learning technique for scientific document classification. Classification of abstracts is done by assigning classes (scientific fields) to abstracts by processing the textual data using a machine learning models. This makes it exponentially easier to sort and manage documents for people/organizations dealing with a lot of information (publishers, bloggers etc.). This project involves analysing abstracts of scientific documents from arXiv. We have accumulated data (given data and additional data collected from online source) and used natural language processing techniques for scientific document classification. Text data processing, building classification models, training the model using the training data, and finally testing the data and model evaluation are the core sub sections for the project. Our machine learning models of choice include, Logistic regression, Support vector machines and XGboost. Our goal is to analyse the text in the abstract (pre-process it first) to predict the label (scientific field) for the abstract. We will go through the pros and cons of the models and explain why one model performs better than the others. Finally, we will share our observations/findings and conclusion.

Objective:

The goal if this project is to create a classification model that can assign labels of subjects to abstracts extracted from arXiv. There are 100 scientific fields and we are using around 130000 (given data plus 100000 randomly selected rows from other sources) records of abstracts and labels to train the models. Our models use processed textual data from the abstracts to classify them into appropriate labels.

Parts of the project:

1. Text pre-processing/Data preparation
2. Generating features
3. Developing classification models
4. Model evaluation and predictions

Text pre-processing/Data preparation:

The date we are given comes from arXiv database of scientific journals. It contains 29638 training samples of abstracts with label column consisting of 100 classes. It is in a csv file which is easily read using pandas read_csv function. We are also given a test dataset containing 7410 testing samples with no label feature.

Data preparation and pre-processing steps:

Adding more data from online source:

Additionally we collected more data from here (https://www.kaggle.com/Cornell-University/arxiv). The additional data we collected came in json format and needed further parsing to extract required features. We noticed that the json data had more than our 100 labels in the test data. So, we only took rows from the new dataset which had the same labels from the training dataset we were given and saved it into a csv file called "extended_dataset.csv". Finally, we combined the training dataset and the additional data (randomly chosen 100000 rows from around 500000) to create a merged dataset of 129638 rows.

Dictionary conversion:

We transformed the new train data and test data into dictionaries where keys are the ids and values are the abstracts.

Further steps:

Moving on, we use various text pre-processing steps to generate features. We used NLTK python library extensively for this part. I have created user defined functions for each of the pre-processing steps which will be used for both training and testing data. The steps are explained below:

- **Sentence segmentation:** Sentence boundary disambiguation (SBD), also known as sentence breaking, is the problem in natural language processing of deciding where sentences begin and end. We used The NLTK's Punkt Sentence Tokenizer to split the abstract text into sentences. It includes a pretrained sentence tokenizer for the English language.
- **Tokenization:** tokenization refers to the division of sentences in paragraphs into words. After experimenting with a few different tokenizers, we decide to go with NLTK RegexpTokenizer. We also experimented with various regexs for the tokenizer and got best results with our custom regex (r"[a-zA-Z$0-9]+(?:[-'_.][a-zA-Z]+)?(?:[-'_.][a-zA-Z]+)?").
- **Generating bigrams:** bigrams are a combination of words which occur together. i.e. fault-tolerance. In our case we only chose bigrams which are happening at least 100 times together and are the top 500 in all the available text data. We are also using the PMI measure and MWETokenizer for tokenizing the bigrams.
- **Removing stop words:** words such as a's, able, about, above are considered stopwords. These words are very common and carry little lexical content. Removing stopwords helps in saving storage space and speeding up processing. We downloaded the English stopwords list from Kevin Bourge's website and read the file into a list. We then used our stopwords removal function to remove the stopwords.
- **Remove most/least frequent tokens:** we have implemented a function to remove most/least frequent tokens, but later decided to not remove rare tokens as it gave us substantially better prediction results.
- **Stemming and lemmatization:** grouping of words such as detail, details and detailed into its base form (detail) is called stemming/lemmatization. The processes of stemming and lemmatization both involve generation of the root form of the word. The difference lies on the fact that stemming process might not generate actual words whereas lemmatization does. We chose to implement both methods in our pre-processing.

Generating features:

Following the pre-processing steps, the converted versions of each abstract is required to be altered into a numeric representation. This conversion makes it so that the data can be used to train and test the classification models. Given a set of documents and a pre-defined list of words appearing in those documents (i.e., a vocabulary), we can compute a vector representation for each document. This vector representation can take one of the following three forms:

- a binary representation,
- an integer count,
- and a float-valued weighted vector.

We have generated features in these ways:

- **Count vectors** converts a collection of textual documents into a matrix containing token counts. Firstly, we initialized CountVectorizer by analyser as word and setting a token pattern. Then we fit the model with the training abstracts. Then finally we generate the count vectors for the test and training data
- **TF-IDF** vectors converts several raw documents into a matrix containing TF-IDF features. We initialized the TfidfVectorizer using analyser as word and setting max features to 50000. Then like the count vectors, we fit the model using the training abstracts then finally generated the TF-IDF vectors for train and test data.
- In NLP, the process of labelling words with their corresponding part-of-speech (POS) tags is known as **POS tagging**. A POS tagger processes a sequence of words and attaches a POS tag to each word based on both its definition and its context. There are many POS taggers available online, such as Sandford POS tagger. We are using the one implemented by NLTK.

Developing Classification models:

For our project, we tested many classification models such as logistic regression, support vector machines, Xgboost, decision tree etc. our final three models are:

1. Logistic regression
2. Support vector machines
3. XGBoost
4. Tuned XGBoost

All these models are implemented using the sklearn module.

Logistic Regression classification model:

It is a binary classification model where conditional probability of one of the two possible realizations of the response variable is assumed to be the same as the linear combination of the input variables which are transformed by the logistic function.

The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It’s an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

Equation → 1 / (1 + e^-value)

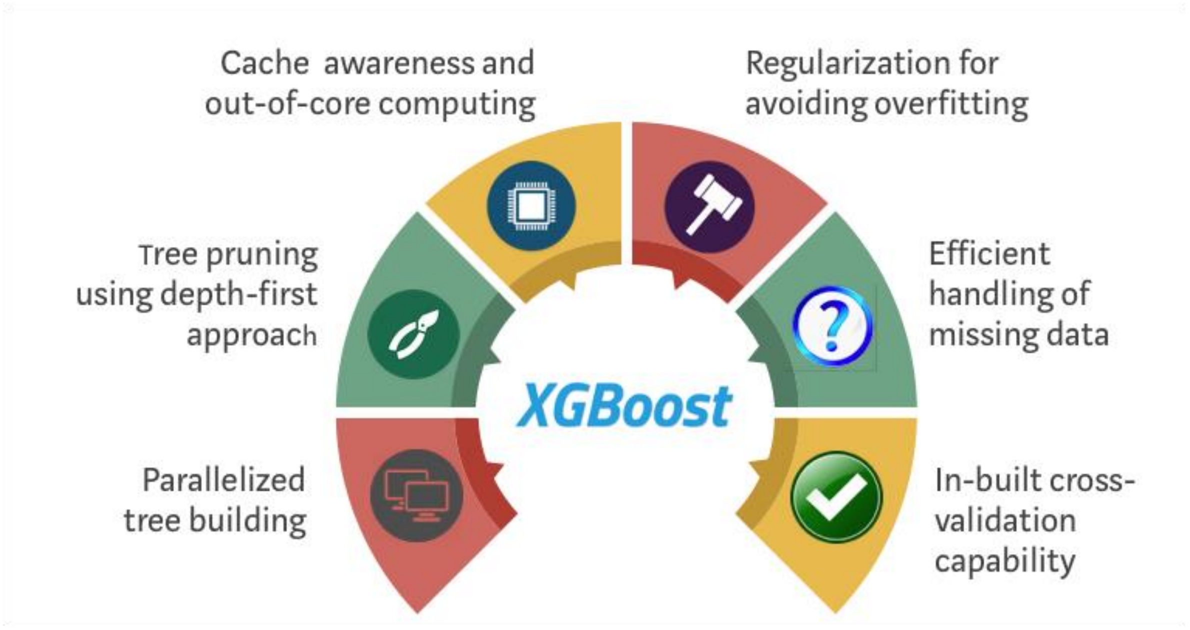
For the most part we have used default parameters for the logistic regression model. We only added max_iter to 1000 and set random_state to 1234 (for reproducibility). LR model always attempts to maximize the likelihood function of any given task. The model was created using the sklearn library. Firstly, we initiate the LogisticRegression function by setting the above parameters and fit the model with the training data (using the TF-IDF vectors). Then we proceed to make predictions, append the predictions to the test data labels and save it to a csv file. We got 56.77 percent accuracy on Kaggle for the logistic regression model.

Support vector machines classification model:

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outlier detection. SVMs are very effective in high dimensional spaces and when the number of dimensions is higher that the samples. They use only a portion of the training data in the decision function, making it memory efficient. We are using the SVC model from sklearn.svm package here. Firstly, we initiated the svc model with a random state then used the training data (TF-IDF vectors) to fit the model. Then we performed predictions on the test data (TF-IDF vector conversions), appended the predictions to test data and saved it to a file. We got 53.12 percent accuracy on Kaggle for the Support vector machines model.

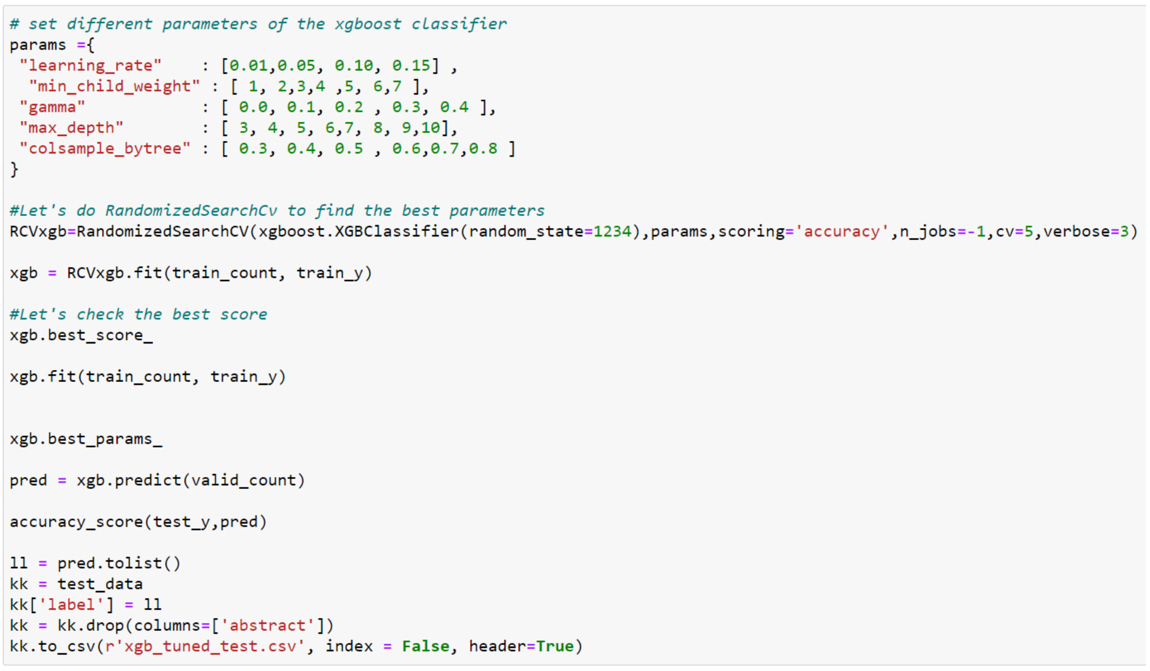
XgBoost classification model:

XgBoost is referred to as Extreme Gradient Boosting model. It is an efficient application of the stochastic gradient boosting ML algorithm. It is also known as gradient boost machines or tree boosting and is a great machine learning technique used on a wide variety of ML problems.


Cache awareness and out-of-core computing
Regularization for avoiding overfitting
Tree pruning using depth-first approach
Efficient handling of missing data
XGBoost
Parallelized tree building
In-built cross-validation capability

XGBoost is an ensemble of decision trees algorithm where new trees repairs faults of those trees that are already belongs to the model. Trees are added until no further improvements can be made to the model. In our case, we used the sklearn library for the implementation of the XGBoost model. Like the SVM model, XGBClassifier was initialized with the same random state and the function was fit using the count vectors training data (count vectors and true labels). Then the predict function was used to predict the labels for the test count vectors. Finally, the predictions were appended into the test data set, abstract column removed and saved to a csv file. For this model, we were able to achieve 50.69% accuracy score after tuning the model.

XG boost tuned



```
# set different parameters of the xgboost classifier
params ={
 "learning_rate"    : [0.01,0.05, 0.10, 0.15] ,
  "min_child_weight" : [ 1, 2,3,4 ,5, 6,7 ],
 "gamma"            : [ 0.0, 0.1, 0.2 , 0.3, 0.4 ],
 "max_depth"        : [ 3, 4, 5, 6,7, 8, 9,10],
 "colsample_bytree" : [ 0.3, 0.4, 0.5 , 0.6,0.7,0.8 ]
}

#Let's do RandomizedSearchCv to find the best parameters
RCVxgb=RandomizedSearchCV(xgboost.XGBClassifier(random_state=1234),params,scoring='accuracy',n_jobs=-1,cv=5,verbose=3)

xgb = RCVxgb.fit(train_count, train_y)

#Let's check the best score
xgb.best_score_

xgb.fit(train_count, train_y)


xgb.best_params_

pred = xgb.predict(valid_count)

accuracy_score(test_y,pred)

ll = pred.tolist()
kk = test_data
kk['label'] = ll
kk = kk.drop(columns=['abstract'])
kk.to_csv(r'xgb_tuned_test.csv', index = False, header=True)
```

To tune the XGBoost model, we set a few parameters such as learning rate, max depth etc. We used the XGBClassifier inside the RandomizedSearchCV function, set the scoring method to be accuracy, utilized all threads, setting 5-fold cross validation as well as verbose of 3. Then we got the best scores and retrained the model utilizing the count vectors of the training data. Then we proceeded to select the best parameters and make the predictions for the labels using the count vectors for the test data.

Model evaluation, discussions and predictions:

For this part we split the merged data (training data) into train and validation splits to test the accuracy. The accuracy_score function was used to test the predicted labels against the train labels. We were constantly getting accuracy score for each model which was around 1 percent lower than what we achieved from the Kaggle website. Out of the 3 models, we achieved the best accuracy score from the Logistic Regression model (56.77%). It was better than both the SVM model and the XGBoost model. The training dataset we have high imbalance which is the probable reason behind logistic regression outperforming SVM model. The XGBoost model was even inferior that the SVM model. As accuracy score is testing data evaluation method, our model of choice is the Logistic Regression model (56.77%). Despite logistic regression being a binary classification model, it was able to do a decent job at predicting multiclass classifications in our case (100 different labels). The SVM model came close to the logistic regression model at 53.12 percent where as the XGBoost model, even after tuning was able to get only 50.69% accuracy score as per Kaggle submission.

After considering all the models and their merits, we chose the Logistic Regression model with TF-IDF feature vectors as inputs and output. The highest accuracy score for this model was 56.77%.

Experimental setting:

To derive the best model, we attempted to create various models. These experimental setups and their accuracy are given below.

Classification model	**Accuracy (%)**
Logistic Regression TF-IDF	**56.77**
Logistic Regression Count Vectors	45.8
SVM Count vectors	44.03
SVM TF-IDF	**53.12**
Decision tree classifier	31.89
Random forest	40.12
XGBoost Count vector	46.38
XGBoost tuned Count vector	**50.69**

During the very first tests, we had tried various pre-processing techniques and feature generation such as removing most/least frequent tokens, using a simpler regex sentence tokenizing etc. we found that not removing most/least frequent tokens substantially increased out model's accuracy score. Then we slowly tweaked a few other parameters to finally achieve our best model which was the Logistic regression model. ***Comparing these accuracy***

results, it is evident that Logistic Regression with TF-IDF vectors had the best result out of all the models and that's why it was chosen as our best model.

Conclusion:

Learning the capabilities of natural language processing with regards to scientific document classification was a great learning experience for us in this project. We learned how to parse json files into data frame format, use various pre-processing techniques, feature generation methods, build and test a multitude of classification models fine tuning them to achieve a better result as well as evaluate and compare different machine learning models. Building a suitable model that can predict a multilabel classification of 100 labels was a daunting task. For now, our best efforts amounted to 56.77% accuracy for the logistic regression model. We hope to keep working on the task after the exams to make it even better.

References:

https://machinelearningmastery.com/logistic-regression-for-machine-learning/

https://mc.ai/xgboost-time-series-for-forecasting-stocks-price/

https://www.kaggle.com/Cornell-University/arxiv