Classifying Authors with Linear SVM Classifier through SGD Optimization

By J. Boogert, A. Fassina, K.W. Tang and Y. Zhang

This paper addresses lead author prediction from a Machine Learning perspective. The aim is to classify unseen papers into predefined labels, represented by the authorId. The content of the paper comprises five sections, namely feature engineering, learning algorithm, hyperparameter tuning, a discussion about performance, and task division.

Feature engineering

Before delving into feature engineering, we conduct preprocessing on both the training and test set. Preprocessing includes extracting the lengths of the "title" and the "abstract" columns and saving them into two new numeric features. All available features are concatenated into a string column named "features". Data cleaning is then performed only on the "features" column via the "clean_text" function. This removes special characters, turns every letter into lowercase, and keeps only lemmas from original words using Lemmatizer (NLTK, n.d.). Feature engineering is then carried out in the pipeline provided by Sklearn, using CountVectorizer and TfidfTransformer. CountVectorizer converts the content of "features" into a matrix of token counts. This results in sparse representation of the counts. The number of features corresponds to the vocabulary size of the data. Finally, TfidfTransformer normalizes the matrix (Scikit-learn, n.d.-a; Scikit-learn, n.d.-b).

Learning algorithm

After transforming features, we train a classifier to predict. When learning the algorithm, we take into account three models. At first, we consider Naïve Bayes and k-Nearest Neighbors, which return baseline validation accuracy of 0.01 and 0.08 respectively. Then, we employ Linear Support Vector Machine (SVM), as it is one of the best algorithms at handling large scale problems and tackling text classification. To this end, the algorithm creates a dot (in 1-D), a line (in 2-D) or a hyperplane (in 3-D and above) to classify. SVM aims to find the 'best' hyperplane so that all the examples are not only assigned to the correct class, but also remaining 'as far away from the hyperplane as possible', which prevents overfitting. As an optimization algorithm, we choose Stochastic Gradient Descent (SGD) over Batch Gradient Descent (BGD). The former uses one randomized training example per iteration, and thus is computationally faster than the latter. It is an optimization algorithm suitable for data with many input features. In scikit-learn, SGDClassifier with loss function set at default ('hinge') provides a linear SVM model (Scikit-learn, n.d.-c). Moreover, SGDClassifier provides the class functions fit and predict, through which we fit the data and perform the prediction. Fit (X, y) fits the linear model with the SGD algorithm. Predict (X) predicts the class labels for samples in X. Combining SVM and SGDClassifier, we reach a baseline validation accuracy of 0.28.

Hyperparameter tuning

Sklearn's pipeline function integrates a big part of feature engineering, model training and classification. Working in parallel with GridSearchCV and its embedded K-Fold (K=3) Cross Validation, the pipeline enables a simpler and more convenient hyperparameter tuning process. In terms of parameters, it turns out the smoothing factor for IDF is not needed, which means terms with - near - zero IDF (words that are very frequent or appearing in all papers) are not used to train the model, which is an alternative automated and less rigid approach to removing, for example, stopwords naively from the corpus. Normalization is done by scaling the sum of squares per row (paper) to 1: it turns out L2 for normalization gives the best accuracy score via GridSearch (Scikit-learn, n.d.-f). Lastly, we apply sublinear scaling to term-frequency. After testing different types of classifiers, SGDClassifier with Linear SVM turns out to be the most suitable for the task. Using gridsearch it turned out the combination of 0.00001 for the learning rate and elastic-net (Scikit-learn, n.d.-c) as a penalty function returned the highest validation accuracy of 0.46, but interestingly only a test accuracy of 0.21. Lowering the learning rate by a factor 10 (0.000001) gave validation scores for elasticnet and L2 of respectively 0.28 and 0.25. Building from the results from our first submission, we thought the model might be overfitting and we chose the second-best model with an increased alpha of 0.0001 (factor 10) and via hypertuning selected the L2 penalty. L2 regularization (ridge) keeps all coefficients, but shrinks them towards zero (StackOverflow & z991, 2016). This adjustment lowered the validation accuracy by just a fraction (0.44), simplified the model, and increased the test accuracy score to just above 0.25. Finally, we thought that we could even use a higher penalty (less overfitting, simpler model). The third best (based on validation accuracy) submitted model has a validation accuracy of just half a percent below the score of the second model. Therefore, we submitted this prediction to CodaLab to see if it would score even better. As this was not the case, we decided to stop tuning. Since there are a lot of single papers in the training set, it was not expected to have a high accuracy overall. Also looking at our fellow students' scores, we did not think we could improve on this with our model. The overall result of these experiments can be seen in Figure 1 and Table 1. Since we didn't have test scores for models with 0.000001 alpha, we didn't include those.

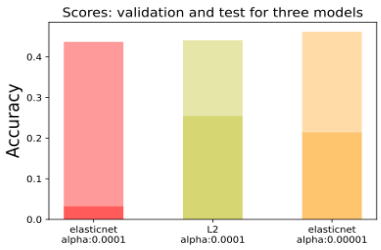

Figure 1. Accuracy scores of the three chosen models on validation set (highest three bars) and test set (lowest three bars).

Model parameters		Accuracy	
Regularization	penalty	Validation	Test
L2	0.0001	0.443	0.254
elasticnet	0.00001	0.460	0.214
elasticnet	0.0001	0.436	0.032

Table 1. Accuracy scores for three different models.

Discussion about performance

Three aspects need to be addressed during preprocessing. Firstly, including the abstract and the title lengths among predictors supports the classifier in extracting author's preferences, potentially enabling better discrimination. Secondly, considering all available features for concatenation when creating the "features" variable stems from the idea that the more the information, the better the prediction. Thirdly, data cleaning does not affect stop words. In fact, these may be useful to capture recurrent patterns in the writing style. As expected, their removal does not improve the accuracy score. As for feature engineering, some observations are due. The CountVectorizer is employed because most algorithms need the sparse representation of the counts to work with text data. No a-priori dictionary is provided, as many words are very domain-specific and are not likely to be found in an external vocabulary. Consequently, the number of features corresponds to the vocabulary size of the data, but in a loose sense. In fact, each resulting column represents the count of either a single word (uni-gram) or a pair of words (bi-gram). Considering bi-grams proves to benefit prediction performance. Indeed, common n-grams convey less information than domain-specific grams, such as "metric". To efficiently account for rarity in n-grams, Tf-idf scales down the impact of n-grams that occur more frequently in a given corpus. Moreover, tuning parameters is more organic than mechanical, as the process builds from insights along the way. For instance, we notice an increase in accuracy when using bi-grams instead of uni- and tri-grams. We did not use an external vocabulary to vectorize our text, because a lot of the words are very domain-specific and most likely not found in an generic external vocabulary. As for the results from hyperparameter tuning, we ascribe the large discrepancy in accuracy scores between validation and test data to overfitting. Since more than 2000 papers have only one author, a better accuracy score on test data cannot be reached unless more training data is provided. Oversampling wouldn't solve the problem in this case.

Task division and progress

To approach this project, we divide the tasks based on our knowledge and capabilities. Jurrien Boogert conducts the supervision over the tasks and the quality control. He also submits our group work in Codalab under the name of *JBoogert*. After getting a basic understanding of the data and programming involved, everyone's task is to create an (attempt of a) NLP model (from preprocessing to classifying authors) to make sure the best parts of the experiments are integrated in our final model. Yin Zhang developed a method to preprocess data with Torchtext, defined a CNN model, and reached a validation accuracy of 0.2855. As preprocessing through Torchtext is rather automatic and not easily interpretable, and since the group hoped to understand NLP in a detailed manner, we decided to progress with the current method. Moreover, Yin Zhang provided insights on defining current title, incorporating the introduction, theorizing the learning and optimization algorithms, restructuring sections for coherence, and helped the group meet the requirements of the project by a detailed investigation of the assignment. On the other hand, Alessandro Fassina and Ka Wai Tang focused on the first prototype of pre-processing for textual data and converting features for the analysis dataset. Further, they contributed to the paper in terms of syntax, lexicon, and style checks. Jurrien Boogert created the initial pipeline with automated preprocessing, vectorization, transforming and a learning algorithm (SGDclassifier). He also incorporated some important features from Alessandro Fassina and Ka Wai Tang's work into the pipeline and found a way of vectorizing and transforming the data into the pipeline using a CountVectorizer and TF-IDF transformer. This made it possible to make hyperparameter tuning more convenient. Additionally, Jurrien Boogert is responsible for finding the best classifier for this model, implementing cross-validation and integrating hyperparameter tuning in the final code. Finally, Jurrien Boogert designed Figure 1 and Table 1.

References

Akshaytheau. (2021, January 31). *Multi class classification using Machine Learning.ipynb*. GitHub. Retrieved December 1, 2022, from https://github.com/akshaytheau/Data-Science/blob/master/Multi+class+classification+using+Machine+Learning.ipynb

NLTK. (n.d.). NLTK :: *nltk.stem.wordnet module*. Retrieved December 1, 2022, from https://www.nltk.org/api/nltk.stem.wordnet.html

Scikit-learn. (n.d.-a). *CountVectorizer*. Retrieved December 1, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html?highlight=countvectorizer

Scikit-learn. (n.d.-b). *sklearn.feature_extraction.text.TfidfTransformer*. Retrieved December 1, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html

Scikit-learn. (n.d.-c). *sklearn.linear_model.SGDClassifier*. Retrieved December 1, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html?highlight=sgdclassifier

Scikit-learn. (n.d.-d). *sklearn.model_selection.GridSearchCV*. Retrieved December 1, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Scikit-learn. (n.d.-e). *sklearn.pipeline.Pipeline*. Retrieved December 1, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Scikit-learn. (n.d.-f). *sklearn.model_selection.GridSearchTextFeatureExtraction.* Retrieved December 1, 2022, from https://scikit-learn.org/stable/auto_examples/model_selection/grid_search_text_feature_extraction.html

StackOverflow, & z991. (2016, January 1). *Regularization parameter and iteration of SGDClassifier in scikit-learn*. Stack Overflow. Retrieved December 1, 2022, from https://stackoverflow.com/questions/34556476/regularization-parameter-and-iteration-of-sgdclassifier-in-scikit-learn