A Model for Language Classification: Implementation of Naive Bayes and Other Classification Algorithms

Team name: Crappy Test Set Predictor
Andrei Chubarau [260581375], andrei.chubarau@mail.mcgill.ca
Guanqing Hu [260556970], guanqing.hu@mail.mcgill.ca
Kamila Mustafina [260748469], kamila.mustafina@mail.mcgill.ca

***Abstract*—Language classification is a widely requested feature in many modern trending applications. This report presents the implementation and testing of several popular language identification algorithms using machine learning, followed by a discussion of their strengths and drawbacks.** [1]

***Index Terms*—Language classification, machine learning, Naive Bayes, K-nearest neighbors**

I. Introduction

The objective of this project is to devise a machine learning algorithm to classify the language of text samples extracted from corpora in five different languages. The training set, including labels, as well as a test set were downloaded from Kaggle's Dialogue Language Classification competition page [2].

The rest of the current paper is structured as follows. In Section II, relevant research and papers are summarized and discussed. Data pre-processing and feature design are presented in Section III. IV describes the implemented solutions, namely the algorithms used. To be more specific, seven text classification algorithms have been implemented, including K-Nearest-Neighbour with K-Means (KNN), Multinomial Naive Bayes (NBM), Random Forests Classifier with Adaboost (AB-RF), Stochastic Gradient Descent Classifier with Adaboost (AB-SGD), and Multi-layer Perceptron (MLP). Bootstrap Aggregation (Bagging) was implemented at the end by combining the predictions from multiple algorithms mentioned above. Note that NBM, KNN, and Bagging have been implemented from scratch, while other algorithms are implemented with the help of the SKLearn package. Related implementation details are described in Section IV, followed by testing and validation results in Section V. Section VI completes the paper with the summary of the achieved results.

II. Related Work

Language identification was established as a task by Gold in 1967 for human subjects. It became popular as a text classification task in 1990s. Nowadays, multi-class text classification becomes more important along with the booming Internet industries that require text identification for many tasks like spam filtering, documents organizing, information searching, etc. Many algorithms for text classification have been developed and researched. [5]

Popular text classification algorithms include KNN, NB, SVM, and many ensemble learning methods. KNN is a non-parametric lazy learner method for text classification. [2] gives a good example of the simplicity of using KNN for text classification tasks. Naive Bayes is notorious for its light weight nature and promising performance for text classification [3]. is a good resource to understand Naive Bayes and its popular extended versions to complete this task. SVM is often the best performing and most robust method for text classification. [4] is one of the most popular paper for solving this task using SVM.

III. Problem Representation

A. Feature Design and Selection

In the current project, most feature preprocessing and selection algorithms are implemented from scratch. Firstly, N-grams are generated for a large dataset of text. Vectorization of text samples follows based on the precomputed vocabulary of N-grams, which are used as features.

We define an N-gram as an N-character long contiguous slice of a string. Generally speaking, a string of length *l* contains one *l* gram, two *l-1* grams, three *l-2* grams, and so on until *l-k=1*. Thus, a given text sample can be decomposed into a set of corresponding N-grams, which can be further used to describe or encode the sample.

Feature generation begins with the training set being decomposed into a large vocabulary of N-grams, top K of which are then used for vector representation of the text samples. An element of a vector at index i corresponds to the count of the i-th N-gram given the precomputed dictionary of K N-grams. For example, in a simple dictionary with two bigrams "ha", "ba", the word "hababa" is represented as the vector [1, 2], where 1 is the count of "ha", and 2 is the count of "ba". Similar logic applies to dictionaries with larger dimensionality.

N-gram extraction from text is a simple process. For instance, for the string "hababa", we have: i) 1-grams: 'a', 'b', 'h' (without redundancy); ii) 2-grams: "ha", "ab", "ba", iii) 3-grams: "hab", "aba", "bab", and so on. The computation simply involves splitting the string onto carefully partitioned substrings. Notice, however, that the longer the length of the N-gram, the more the structural specifics of the sample are preserved. In such a way, computing combinations of various lengths of N-grams indirectly extracts linguistic particularities from textual data.

In our implementation, the result feature analysis is a large Python dictionary hashing N-gram value (string) to its number

[1] All of the implemented Python code can be found at the following GitHub repository: https://github.com/ch-andrei/LangClassifiers-P2.

[2] https://www.kaggle.com/c/comp551-language-classification/data [1]

of occurrences for each language (an array with length equal to the number of considered languages). Note that the computation of this large dictionary is done prior to the training, and as such is part of the data analysis.

The best features (N-grams) are selected based on an importance heuristic which takes into account the N-gram's frequency and its interlingual uniqueness. The heuristic provides insight into how useful an N-gram may be for distinguishing between the entries in the dataset. For true language classification on real text samples, longer and more unique N-grams would provide more information for classifying the language of the sample. If multiple N-grams are available but only K of these will be kept for the vectorization stage, the N-grams that carry the most weight for classification purposes should be used.

The heuristic's frequency factor for a given N-gram is computed simply as its proportion to the total number of counted N-grams in a language. This is easy once the large N-gram dictionary is computed, since all the relevant counts are easily accessible.

The uniqueness of an N-gram is estimated over its distribution across the considered languages; an n-gram which is evenly distributed over all languages is considered less unique than one which occurs in fewer languages. The uniqueness is estimated by computing the magnitude of the cosine of the angle to the least unique vector. For example, for a two-language classification problem, vector [1, 1] is the least unique vector (i.e. the N-gram occurs equally many times in language A as in language B), while vectors [1, 0], and [0, 1] are the most unique vectors. Similar logic applies to larger dimensions of the classification classes. The main advantage of using the uniqueness measure is to nudge the scores of N-grams to prioritize more unique N-grams over those that are simply more frequent.

The results of selecting best features using the heuristic in comparison to simply using N-gram frequencies can be observed in Figure 2 and 3 in the Appendix A. The difference is that some N-grams that might otherwise be discarded are prioritized due to their higher heuristic value; the two figures clearly show how some N-grams change their order of importance, pushing more unique N-grams to the front of the list.

Once the best features are selected, vectorization of text can be carried out. It must be pointed out that limiting the length of the vectors is important in order to minimize the computational costs associated with encoding, decoding, and manipulating the vectors. This aspect puts emphasis on the importance of picking the features which carry the most information, which is the main justification for using the heuristic described above.

B. Training Set

While the publicly available training set was used for the original evaluation of the language profiles, we also created a separate special training set based on the original data. This was done for three reasons. Firstly, the provided test set is very different from the given train set; secondly, we wanted to have a convenient way of locally testing our classifier implementations without having to rely on Kaggle to score our submissions; lastly, because we wanted to verify the effect of training on the generated set in regards to classification accuracy.

The generated dataset attempts to mimic the available test set: it is essentially a combination of randomly sampled characters, generated using the character distributions available in the original training data. Since we thoroughly analyzed the training data in our feature selection stage, we can easily extrapolate and generate as many fake sets as we want.

Summary tables for the original training set (Training Set One) and the fake training set (Training Set Two) are shown in Table I and Table II.

TABLE I: Summary of Training Set One

Class	Language	Number of Samples	Mean of Characters
1	Slovak	14167	52.64
2	French	141283	49.81
3	Spanish	69974	44.63
4	German	37014	49.60
5	Polish	14079	41.47
	Total	276517	

TABLE II: Summary of Training Set Two

Class	Language	Number of Samples	Mean of Characters
1	Slovak	11105	35.84
2	French	105558	35.60
3	Spanish	47061	35.48
4	German	27950	35.37
5	Polish	8326	36.35
	Total	200000	

C. Data Cleaning

The original training set has multiple incorrectly classified or badly formated samples that may hurt the training of our model. Some cleaning was applied before putting the training set into use. As a result, 2867 empty samples and 148 purely numeric samples were discarded. Table III presents a summary of cleaned training set one. The rationale behind removing empty samples is self explanatory: they do not contribute to training. Numerical samples, on the other hand, were a subject of discussion. In the end, it was decided that numbers should not be used to classify the language of a sample, since, in theory, numbers are common for all languages.

TABLE III: Summary of Cleaned Training Set One

Class	Language	Number of Samples	Mean of Characters
1	Slovak	14163	52.64
2	French	140708	49.83
3	Spanish	68616	44.66
4	German	36151	49.64
5	Polish	14044	41.48
	Total	273682	

IV. Algorithm Selection and Implementation

The implemented solution essentially consists of three stages. Firstly, the train-set is thoroughly analyzed to produce a dictionary of best features which will later be used to encode sentences as vectors. Following that, a classifier is selected and trained on the vectorized training set. Lastly, the test-set is read and predicts are computed using the trained classifier.

Table IV summarizes the results achieved by the implemented classification algorithms, as given by our local validation and Kaggle's scoring mechanism. While we attempted to classify the language of sentences using multiple approaches, the most effective and surprisingly the simplest solution was given by our implementation of the Naive Bayes (NB) classifier. Other, often more complex, models fell short of the performance given by NB. The following classifier types were used, ordered by classification accuracy (* indicates usage of SKLearn's implementation, lack thereof indicates our own implementation): Multinomial Naive Bayes (NBM), Bagging via voting, Adaboost with Random Forests (AB-RF*), Multinomial Naive Bayes (NBM*), Adaboost with Stochastic Gradient Descent (AB-SGD*), Multi-layer Perceptron (MLP*), and K-nearest Neighbors (KNN). Note that some results were not scored by Kaggle and hence have "N/A" as their testset score.

TABLE IV: Performance of Algorithms Summary

Algorithm	Validation Accuracy (%)	Test Accuracy (%)
NBM	79.511	78.729
Bagging	77.107	78.572
MLP*	74.295	77.269
NBM*	75.591	77.176
AB-RF*	75,766	N/A
RF*	75.583	N/A
AB-SGD*	73.905	N/A
kNN	61	55.105

A. Feature Selection

While our original intention was to use N-grams with various length, we soon observed that using N-gram lengths higher than 1 is in detriment to classification accuracy for all used classifier types for the provided test-set. This relationship is shown in Figure 1 in which Naive Bayes classifier is used. As a result, it was unanimously decided to only use 1-character long N-grams, which in a sense, degenerates the problem of text classification to simply counting frequencies of characters for each language.

The aforementioned is a sad observation, because when tested on real textual samples, our classifiers achieved up to 97% accuracy (Random Forest Classifier with 1-8 long N-grams), which is quite impressive for a language identification algorithm. Unfortunately, this project is not about language classification of real textual samples, but rather a crappy set of randomly generated characters, hence the choice of our displeased team's name.

Note that computational cost and memory consumption increase tremendously when a wide range of N-gram lengths are queried; we used multiprocessing to speed up the computation and implemented training data processing in batches to minimize the memory footprint of the processing algorithm.

B. Validation Set

As was previously mentioned, the original test-set contains text samples in the form of series of randomly drawn characters given the original character probability distributions as given by the train-set. The original train-set, however, contains samples of real conversations. Although the train and the test sets are related through probabilistic distributions of characters, they are very much unlike each other.

The dissimilarity between the train and the test set implies that the validation of our classifier algorithms cannot be directly run on a subset of the train-set. In fact, our results show that the classification accuracy on a subset of the train-set does not correlate with the final accuracy on the test set. With that in mind, our local testing and validation is run on a locally generated test set, for which the language of each sample is known. The generated test-set, the creation of which was described in Section III-B, mimics the structure of the original test-set as closely as possible.

Validation on the generated set proved to be a very effective tool at estimating our final accuracy as given by Kaggle's scoring mechanism. In addition, we also learned that the the length of the queried text sample affects prediction accuracy: longer inputs are easier to classify. This is due to the fact that longer sentences are more informative in regards to their linguistic characteristics, and hence are more distinguishable by the classifiers.

C. K-Nearest Neighbors

KNN is a simple classification method for language classification. [7] One of its main advantages is the absence of the training phase; however, it also requires longer classification times due to the necessity to process the entire set of training samples to identify the elements of the training set that are the closest to the language samples present in the testing phase. In simplest terms, kNN classification is based on calculating the Euclidean distances between the test set sample and the train set members:

$$distance = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{1}$$

In order to compute the distance between the vectors, every line of the training set was transformed into a vector with mapped N-grams from the previously generated dictionary of the most prominent features; the same vectorization process was used to process the testing set. Only 1 N-grams were used in the construction of the set.

D. Naive Bayes

Naive Bayes is a simple and commonly used algorithm in language classification. In the current project, the implementation of Naive Bayes is mainly based on two papers

[3] and [5]. The first paper discusses how to improve Naive Bayes for text classification, while the second paper presents a detailed examination of Multinomial Naive Bayes under different feature conditions.

Multinomial Naive Bayes has been used in this project with Laplace smoothing. Multinomial distribution has been selected for Naive Bayes because it can model a feature in the text sample that shows up several times using P^f, where P is the probability of the feature and f is the number of times the feature appears. This is exactly what the N-gram dictionary computed during the feature evaluation stage provides us with, and the data structure has been optimized with that goal in mind. The final prediction of a sample corresponds to the class with the highest probability as calculated using Equation 2.

$$prediction = argmax_{l_i \in L} P(l_i) \prod_{j=1}^{|V|} \frac{P(t_j|l_i)^{f_{D,t_j}}}{f_{D,t_j}!} \tag{2}$$

$$P(t|l_i) = \frac{1 + \sum_{k=1}^{|D|} f_{k,t} P(l_i, D_k)}{|V| + \sum_{j=1}^{|V|} \sum_{k=1}^{|D|} f_{k,t_j} P(l_i, D_k)} \tag{3}$$

Where l_i is one of the language from a language set L, $P(l_i)$ is the probability of the i-th language among the training samples, V is the set of all features and t_j is the j^th term in V, f_{D,t_j} is the frequency of j-th term in the sample D.

When language is l_i, the probability of t_j, $P(t_j|l_i)$, can be found using Equation 3. Calculating this probability for each term in feature set corresponding to each language is actually the model training step of Naive Bayes. Instead of using Laplace Smoothing, the MAP estimation used in Equation 3 works slightly better when using the number of features, |V|, as denominator.

This project implemented Naive Bayes from scratch along with feature extraction tools that converts raw text samples into N-gram vectors. A lot of tunings has been done around the feature set, as was previously described. Multiple features spaces were separately tested, namely the size of length of N-grams, and their ordering via scaling by importance heuristic. The number of features being considered for each language was also tuned to get the best result on validation and test set.

Finally, note that two slightly different versions of Naive Bayes were implemented. One was an early implementation which directly uses the training samples to compute relevant probabilities, while the second is a refactored and a better optimized version which utilizes a common interface with the other implemented classifiers. The second version relies on the dictionary of N-grams to be precomputed, including all counts and statistics. This is an effective approach, because this step is the same for all classifiers, and thus is shared. It is significantly faster, and offers slightly better prediction accuracy.

E. SKLearn Package

In our exploration of the solution space for the current problem, we attempted multiple classification techniques offered by the Scikit-Learn package. The used methods and results are listed in table IV.

Since these methods did not achieve a higher accuracy than our base classifier type (Naive Bayes) and since their usage is not the priority of the current project, no further details will be hereby provided. Please note, nonetheless, that a lot of effort was spent on exploring the possible configurations.

V. Testing and Validation

The current section pertains to detailed analysis of the achieved results.

A. K-Nearest Neighbors

The main challenges associated with the implementation of kNN classification on the provided text classification problems were the following:

1.) Taxing computation. Since traditional kNN algorithm requires computing distances to each member of the train set to identify the closest neighbors to the sample of interest, **kmeans** clustering was implemented in the pre-processing step to decrease the number of features in the training set. To achieve that goal, the lines in the training set were sorted into groups according to language classes to generate an equal number of clusters from each group. Next, the cluster centers were used to construct a new training set for kNN classification. A correlation was observed between accuracy of the classification of the train set and the number of generated clusters. As an added benefit, the generation of the train set from clusters allowed to alleviate the bias arising from a variation in the number of lines attributed to each class in the original dataset.

2.) Equal scores for various classes. kNN relies on identifying the class that has the highest occurrence throughout the training samples that are the closest to the target vector; however, its design does not include into account the variance between the distances from the target among the chosen k number of neighbors. [6] Hence, weighted kNN was implemented: each vector in the nearest neighbor group was assigned a weight parameter: $weight = \frac{1}{distance}$.

Therefore, in the case of equal counts between multiple classes, the priority was given to the vectors that were closer to the target

For testing, the main parameters for optimization were the number of nearest neighbors chosen from the training set elements and the number of clusters formed for each language set. For the construction of N-gram dictionary, generated test set with 500000 lines was utilized, after the best 559 features were selected. For validation, a series of tests with 50 generated clusters per language was conducted with different parameters:

The values in brackets represent accuracy scores obtained without weighing the distances between training and target vectors. The split of the native training set into training and validation sets was performed with scikit-learn train test split method. The preordering of training vectors into clusters was implemented with scikit-learn KMeans method. Various k values were tested: values below 4 gave a more significant error rate, and values abore 12 did not provide significant accuracy boost with regards to the increase in the computation

TABLE V: Performance of the wtkNN algorithm with 250 clusters

Train set (n)	Validation set (n)	k	Accuracy (%)
Generated (100000)	Generated (5000)	4	0.47 [0.42]
		8	0.42
Provided train (273750)	Provided train (2766)	4	0.64 [0.63]
		8	0.65
		12	0.64

time. While the overall accuracy of the prediction with such a low number of clusters did not compare favorably to the results obtained with classification with Naive Bayes, it could be justified by the variation in the validation success between the generated test sets, consisting of separated characters as in the provided test, and the training sets, which consist of words and sentences.

B. Naive Bayes

One main factor that influences the performance of Naive Bayes is the feature set because Naive Bayes classifier predicts based on the probability for each selected feature. During this project, two hyper-parameters were tuned to generate different feature sets. One is the maximum N-gram length N, the other is the number of top features to keep (ranked by feature importance heuristic). N-grams used in experiments were case-insensitive following the nature of test set (all characters were converted to lowercase).

1-gram and 1-3 grams dictionaries were tested with maximum feature set size ranging from 20 to 16770. The validation and test results are shown in Figure 1. These results are slightly different from the final results in Section IV because these experiments used only a part of the training set One to save computing time (10000 samples), but the results produced are representative enough and provide good feature suggestions for the real training. The test set used is the locally generated version of the real test set as described in III-B.

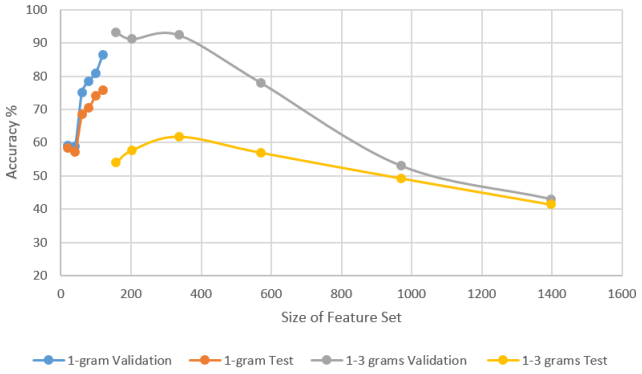

Fig. 1: Naive Bayes Feature Tuning Results

As can be seen in Figure 1, the test set scores are better for 1-gram over 1-3 grams. It makes a lot of sense considering the generation method of the test set is character sampling, where the relation between characters has no meaning. 2-3 grams or higher order grams are able to model the relation between substring characters better, including beginning of words, end of words, etc. The modeling by 1-3 grams with richer information actually helps to get better performance on the validation set (93%) than 1-gram (86%). The full set of 1-gram features has been chosen as the final feature set for Naive Bayes training to maximize both validation and testing performance. Training with generated and native training sets was attempted; the generated training set gives slightly better performance.

Tests have also been done on Laplace smoothing parameters. However, there is no big difference noticed when tuning the smoothing parameter from 2 to the number of features.

VI. Discussion

A disadvantage of vectorizing textual data using the form defined by the current project is that, no matter the size of the vector, some information may still be lost. Consider a vector V of length K which encodes the counts of K different N-grams, and a sentence S that produces a list L of N-grams such that there exists an N-gram G in L that does not appear in V. Then, the information given by G cannot be encoded by V and hence will be lost. In the worst case, G is the only computed N-gram for S, in which case, for prediction, the predict query for S is degenerate, and for training, the sentence S offers no information as all entries in its vector form are zero (this, in fact, may pollute the training).

Among other issues, one of the main challenges of representing sizable amounts of textual data is the storage requirements of large dictionaries. In the case of the current project, if a wide range of n-grams is used (for example, using n=1,2...10), then the entire space of generated n-grams amounts to tens of thousands of samples, which requires excessive storage capacity. While our final approach was to use a particularly well suited configuration based on Python dictionaries, which employ hashes to access and store n-grams, we initially started with a lot less-optimized data structures, which were huge bottleneck and memory sinks. Our final implementation's memory footprint is minimized while maximizing access time and compute capabilities. These properties are crucial for efficient training and prediction of the classifier and as such were a high priority in the current project.

VII. Statement of Contributions

We hereby state that all the work presented in this report is that of the authors.

Andrei Chubarau explored feature selection and generation, implemented Naive Bayes algorithm, and added support for the SKLearn classifiers. Guanqing Hu worked on our implementation of Naive Bayes classifier (original version), as well as feature exploration and data cleaning. Kamila Mustafina worked on the implementation and tuning of the k-NN classifier with weight distribution of neighbors and kmeans clustering with SKLearn package. All the authors contributed to the final report.

REFERENCES

[1] M. Smith, S. Thakur, et al. *COMP-551: Applied Machine Learning Mini-project 2: Language Classification* (2017).

[2] P. Soucy, G.W. Mineau (2001). *A simple KNN algorithm for text categorization*.Proceedings 2001 IEEE International Conference on Data Mining, San Jose, CA, 2001, pp. 647-648. doi: 10.1109/ICDM.2001.989592

[3] J. Rennie, (2001). *Improving Multi-class Text Classification with Naive Bayes*. [online] Available at: http://qwone.com/ jason/papers/sm-thesis.pdf [Accessed 22 Oct. 2017].

[4] T. Joachims (1998). *Text Categorization with Support Vector Machines: Learning with Many Relevant Features*. Machine Learning: ECML-98: 10th European Conference on Machine Learning Chemnitz, Germany, April 21âĂŞ23, 1998 Proceedings

[5] T. Baldwin, M. Lui, (2010). *Language Identification: The Long and the Short of the Matter*. Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the ACL: 229âĂŞ237.

[6] K. Hechenbichler and K. Schliep, (2004). *Weighted k-Nearest-Neighbor Techniques and Ordinal Classification*. Collaborative Research Center 386, Discussion Paper 399

[7] *A Complete Guide to K-Nearest-Neighbors with Applications in Python and R*.[online] Available at: https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor [Accessed 22 Oct. 2017].

APPENDIX

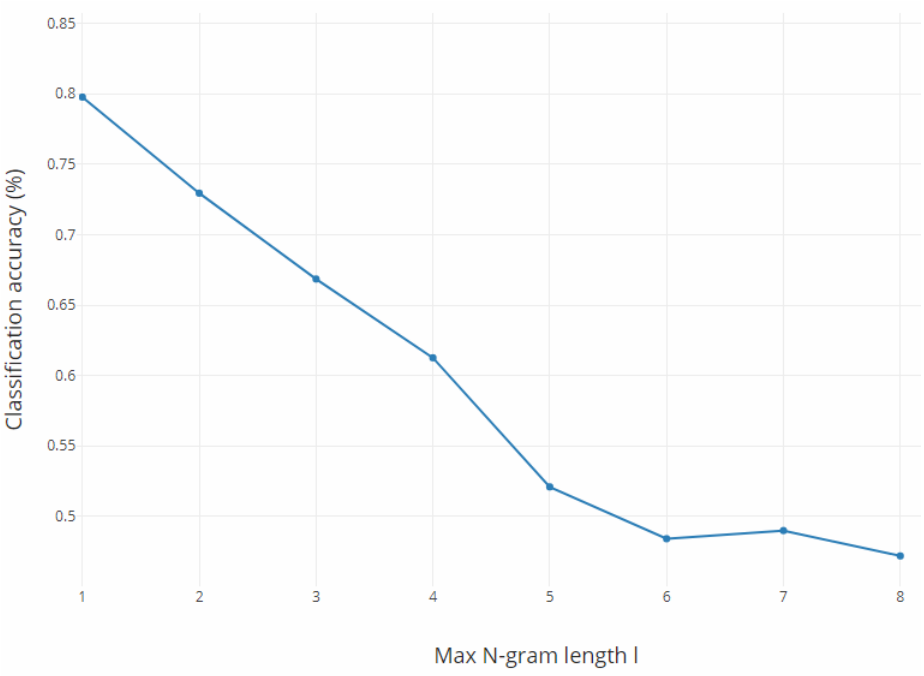

Fig. 1: Naive Bayes Accuracy as a function of maximum N-gram length

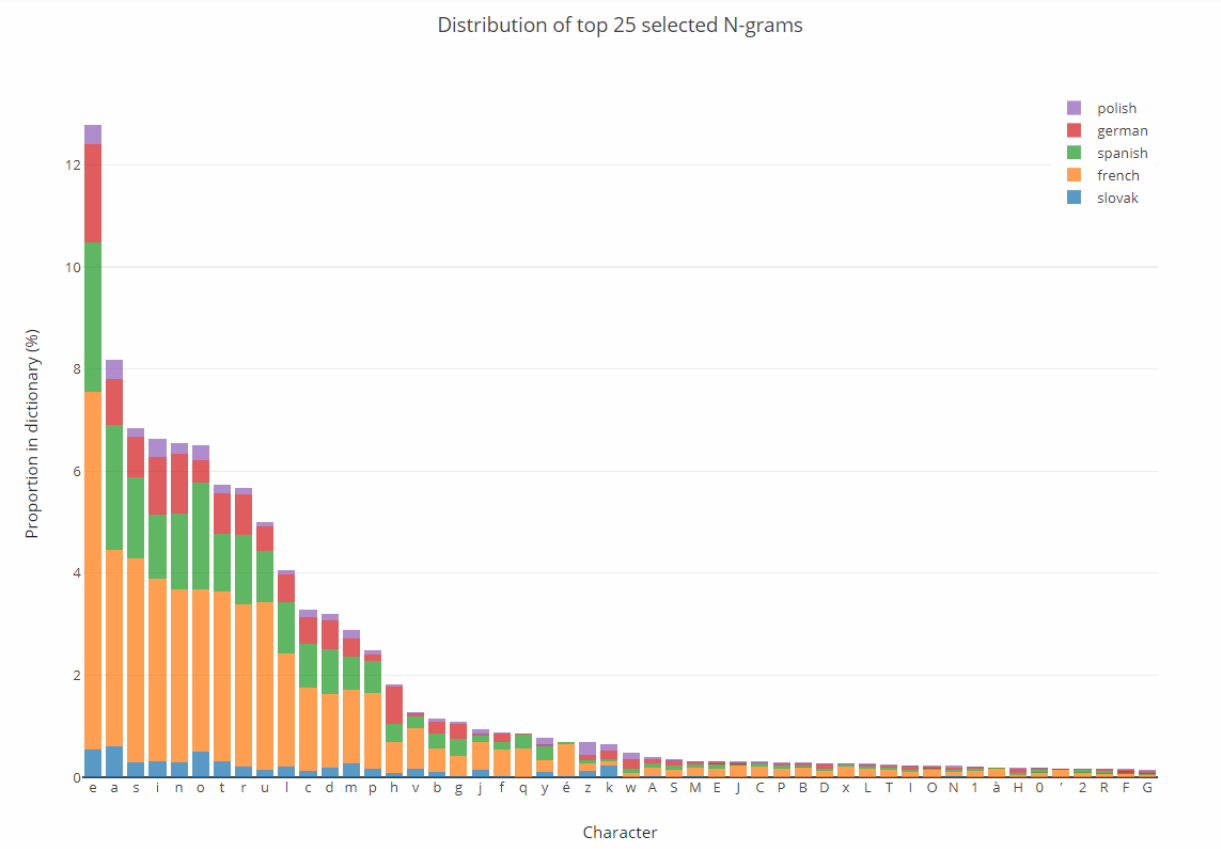

Fig. 2: Top N-gram features ordered by frequency

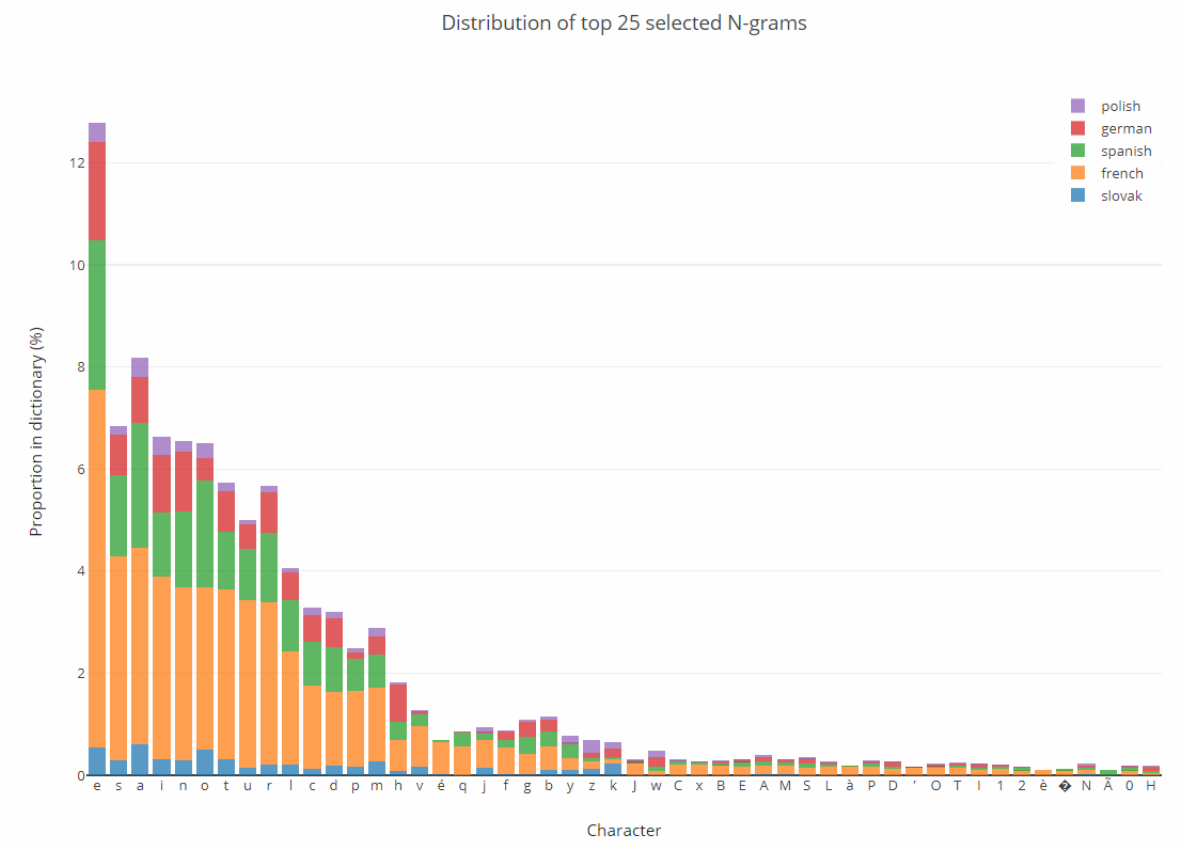

Fig. 3: Top N-gram features ordered by importance heuristic