Determining Language from a Probabilistic Representation of Characters

Kaggle Teamname: Lazy Sloths

Matthew Cooke
matthew.cooke2@mail.mcgill.ca
260553365

Yimiao Ou
yimiao.ou@mail.mcgill.ca
260144236

Lino Toran Jenner
lino.toranjenner@mail.mcgill.ca
260793554

I. Introduction

In machine learning, we use large scale data-sets to train classifier in an attempt to predict the outcomes of related novel events. In this project, we were given a list of over 200,000 data sample of text classified in one of 5 languages: Slovak, German, French, Polish and Spanish. We were also given unclassified strings of random characters probabilistically sampled from these languages. Our goal was to train different, manually implemented machine learning classifiers to solve this language classification task with the highest possible accuracy. We trained 5 different classifiers: Two Naive Bayes, k-Nearest Neighbour, Support Vector Machine, and a Decision Tree. Our best classification accuracy on the public leaderboards, using Naive Bayes with Laplace smoothing was 78.96%.

II. Related Work

The classification of text has been a problem in machine learning for many years. As one of the classic topic in natural language processing, text classification is widely used in areas like spam email filtering [1], document categorization [2], sentiment analysis [3] and web searching [4]. Text classification task normally involves the following steps: 1) document preprocessing, 2) feature extraction and selection, 3) model selection and training, 4) classifier testing. These different steps have been the major focuses of many researches [5]–[7]. For model selection and training, the best working approaches feature a combination of classifiers such as Naive Bayes, Decision Tree, Neural Network and Support Vector Machine [8]–[10]. However, the scope of text classification is vast; there are single-label or multi-label approaches, Hard vs Ranking Categorization, and other design decision to consider. [7] There are hundreds of applications of text classification, one of which is automatic language detection, as described in this report. Other approaches to language classification focus mainly on the occurrence of words [11], however this can be generalized relatively well to individual characters. Recently, character-level text classification has been the focus of many studies [12]–[14].

III. Problem Representation

The task of this project was language classification. While the data of the training set consisted of full sentences, the data of the test set did not. Instead, the test set contained only characters (instead of words) that have been sampled from sentences using a standard probabilistic distribution. These characters were sampled from sentences of the same type as in the training set. Because of this sampling process, character transitions and similar other aspects of language structure were lost. To take this into consideration, our feature selection took place on a character level. After removing space and end-of-line characters from the training set, we used the SciPy [15] *CountVectorizer* which creates a vector representing how often each character occurs in a sentence. Using this approach we get a vector consisting of 598 entries (one for each character that occurred more than once in the whole training set) for each training sentence. We made sure to also account for numbers and punctuation marks as these seemed to provide meaningful information (see discussion).

Once these vectors are built, the data is ready to be classified.

IV. Algorithm selection and implementation (for each of the categories above)

A. Feature selection

After reading in the sentences we shuffled them to avoid any possible biases they might have. We then created the feature representations as described above. Once that was done, we split the 276517 sentences into two parts: the first 250000 made up the training set and the last 26517 sentences made up our development set on which we later calculated the validation error and tuned our hyperparameters. We used these training and development sets for all three machine learning classifiers we used.

B. Part 1 - Naive Bayes

For part 1 of the project, we decided to implement Naive Bayes algorithm. Naive Bayes uses the Bayes rule to output a classification for a sample with We had to calculate the We first counted the occurences of each of the classes in the training set and devided by the total number of sentences to estimate the priors of each of the classes (P(y)). We then counted how

often each character character appeared in sentences of each particular class. We divided this number by the total amount of characters per class to get an estimate of how representative each character is for each class (P(Xly)). By doing this, we got a matrix with the two dimensions class and character. In each cell we store the probability of finding a certain character if the sentence is from this class.

To prevent the numbers of getting too small and creating underflow problems, as well as to facilitate computation instead of storing the probabilites directly, we store natural logarithm of both of these probabilities.

To classify a sentence, we first create its feature vector. We then multiply this feature vector (using the dot product) with our probability matrix to effectivly sum up all the individual character probabilites for all of the classes, returning one sum per class. After this is done, we add the logarithmized class priors to these sums to get the final probabilities. We then find the biggest of the five values and return the corresponding class.

Since characters that occur in the development or test sets might have not been obeserved in the training set for certain classes we also apply laplace smoothing by adding one occurence to each character and 598 occurances (in our case) to the total character occurances per class before calculating the probabilities.

C. Part 2 - Decision Tree

In part 2 of the project, we were tasked with implementing a non-linear classifier. We implemented a 1-nearest neighbour classifier by adding each sentence to a vector and mapping these vectors in their respective 598 dimensional space. We would then compare all of the test sentences to each of the training vectors. The closest training vector would assign its class to the test sentence. Although functional, the way it was coded, this method was much too computationally intensive, computing distance from 200,000+ training vectors for all 100,000+ test vectors and we were not able to let it finish executing. We therefore decided to refocus on a decision tree based approach.

For the second part of the project we wanted to see if changing our feature selection method would still lead to good results. We therefore didn't use the CountVectorizer but a manually implemented feature selection process for our decision tree. While still focusing on characters, we effectively described the training sentences by the sets of characters they contained. That is to say we looked only at which characters appear in a sentence and not how often they occur. For this classifier, we also removed numeric characters.

As a critical part of machine learning, feature selection is always performed before training by most algorithms to improve the quality and efficiency of modeling.

In contrast, decision tree classifiers bypass this preprocess step by incorporating a built-in feature selection mechanism in the training phase.

Decision trees are built in a top-down fashion. At each node they consider all features and select the feature that best splits the node into purest possible classes, which means splitting the node with the feature that achieves the highest information gain. Therefore, the features used to classify languages in our decision tree are all the different symbols appeared in the training dataset. Since numbers provide no useful information for language prediction we remove them from the set of features before training the tree. Totally, there are 665 unique features for splitting a decision tree.

The training dataset was preprocessed as discussed above, resulting in a list of symbols without repeat. The correct class label was then attached to the end of the list. We chose a simple validation set approach and used our training set to train the tree and the development set to calculate our validation accuracy.

In our decision tree model, the structure of the tree is recorded as a dictionary where each node(key of the dictionary) denotes the best feature that splits data at that node. Each node can either have a leaf (class label) or a subtree as a child.

The implementation details are as follows. Given a set of labeled training dataset: 1. When all samples in the dataset belong to the same class create a leaf with the class label and exit. Otherwise, do following procedures: 2. Iterate through the whole set of features, each time split the samples of the dataset into two subsets based on the presence or absence of each feature in the samples. Calculate the entropies and choose the feature that gives the highest information gain as the best feature to split data on. 3. Split the dataset according to the best feature. Delete this best feature from the features set before executing the next run of data splitting. 4. Calculate the number of subtrees for the best feature(key of the subtree) based on whether the best feature is present in all samples(one subtree with key = 1) or absent from all samples(one subtree with key = 0) or is present in some samples and absent from others(two subtrees with key = 1 and key = 0 respectively). Recursively repeat steps 2 to 4 on each subset of the training dataset will eventually produce a full decision tree.

Since the training dataset is very large and contains hundreds of features, generating a decision tree with a regular laptop will take approximately 3 hours. It thus will be very time-consuming to generate a new tree for prediction every time. Therefore, we stored tree dictionary in the hard-drive using the python pickle API which implements binary protocols for serializing(storing) a Python object structure and de-serializing (loading) the object when we need it for new sample prediction. To show that the training process of the decision tree works, we also implemented a demonstration mode that uses a significantly reduced amount of training samples to build the tree. Details on how to run our program in demonstration mode can be found in the README file. Alternatively we also upload a pretrained decision tree that can also be used.

Classifiers	training set accuracy (%)	development set accuracy (%)	kaggle test set accuracy (%)
Manual Naive Bayes (part 1)	86.77	86.70	78.88
Manual Decision Tree (part 2)	98.83	74.60	63.79
scikit Naive Bayes (alpha = 0.2)	86.80	86.72	78.93

TABLE 1. Performance of our implemented classifiers on training data, development set and in the kaggle competition. Accuracy measured in %.

For classification of a new sample, we first create the set of characters to get the appropriate feature vector.We then start at the root of the decision tree, traverse down the tree until reaching a leaf node and assign the class label of that leaf to the new sample. At each node of the tree, choosing which branch to traverse down depends on whether the sample contains the feature that splits that node. A key = 1 implies the feature is inside the new sample. After iteratively predicting the class label for each sample in the test dataset, we calculate the error rates and save the predicted classes of the test set.

D. Part 3 - scikit Naive Bayes

To try how the scikit Naive Bayes compares to our implementation we tried it out and played with the smoothing. One thing we tried with good results was setting the soothing a bit lower than laplace (alpha = 0.2).

V. Testing and Validation

We calculated training accuracy and accuracy on a development set (part of the training data).

Improvement of results could generally be seen mostly by changing the feature selection process. Our first implementation of feature selection ignored any punctuation in the sentences. Our best classifier using this strategy was a Naive Bayes algorithm with Laplace smoothing (training accuracy (ta) 85.45%, development set accuracy (da) 85.49%, public kaggle leaderboard (ka) 77.16%). After tuning our feature selection process the scores of the best performing algorithm on kaggle (Naive Bayes with laplace smoothing) improved to: ta=86.80%, da=86.72%, ka=78.93% (Table 1).

VI. Discussion

For the algorithms we described in this report, each has its own advantage and disadvantage. As a popular text classifier, Naive Bayes is simple to implement and converges quickly. It performs quite well, doesn't require a large dataset for learning to begin and doesn't overfit easily [16]. However, it is based on the conditional independence assumption and is sensitive to parameter optimization. Therefore, choosing which features to train the classifier affect the model performance. Similar to Naive Bayes, KNN is also a simple algorithm to implement and use. It is robust to noisy training data, doesn't need a training phase and learn complex model easily. However, it requires lots of memory to store all the training data and its test phase is slow, which may the reason that caused the failure of our KNN classifier in this project to implement the text classification task efficiently. Furthermore, KNN is hard to apply to high-dimensional data [17]. SVM is a very accurate classifier which uses "kernel trick" and is defined by convex optimization problem, therefore it can be efficiently implemented. It is not prone to overfitting and it does a good job at handling missing data. However, SVM is a binary classifier and could not do multi-class classification. It also requires lost of memory storage for the data [18]. There are several advantages of using decision trees for solving classification problems [19]. Firstly, decision trees automatically perform feature selection where the most important variables within the dataset are the features that used to split the nodes on the top of the tree. Secondly, decision trees are non-linear classifiers, therefore we can use them to fit models without any assumption of linearity of the dataset. Thirdly, decision trees don't require parameter normalization because the "distance" between samples is not used for the training process. Fourthly, decision trees can deal with noisy and incomplete data and can be used in ensemble methods to reduce the variance caused by randomization. Fifthly and most importantly, the learned function and the decision tree itself are easy to interpret and explain. However, decision trees also have some disadvantages: Firstly, the tree output is very sensitive to alterations in the training data. Because features of each sample are used to train the tree, a small change in the input may result in a drastically different tree. Secondly, building a large tree with few hundreds of features and hundred thousand of training data will take lots of time and efforts since information gains are needed to calculate for of each possible split at each node. Thirdly, decision trees are easily overfit. Although this can be negated by methods like validation test and pruning, the application of these methods to a large tree with hundreds of nodes will be very time-consuming. As can be seen in the results, it appears that our implemented decision tree did not do as well as the other classifiers, namely naive bayes. This has probably to with our feature selection since we lose information by only using the sets of used characters. Also, since the training error is really low, it seems that this model overfit to the training data. This could be avoided in a later project by including more rigorous pruning of the tree.

Fourthly, the decision tree prediction accuracy is low, which may be improved by applying ensemble methods such as random forest at the loss of interpretability. However, random forest is not suitable for fitting dataset with missing variables, especially for our training data lots of the samples contain less than 0.1 of the total variables.

VII. Statement of Contributions

All members of our group worked hard on this project. The whole group helped write the report, make major design decisions, and write classification code. Specifically, Lino wrote the Naive Bayes classifiers (parts 1 and 3), worked on bringing all the code of the team together, the usability of the python code and parts of the report. Matthew worked particularily on making kNN and SVM classifiers (which weren't used in the final submission), doing code cleanup and refactoring, and the report: writing, doing research, and collecting references. Yimiao wrote the decision tree classifier (part2) as well as parts of the report.

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

References

[1] T. A. Meyer and B. Whateley, "Spambayes: Effective open-source, bayesian based, email classification system.," in *CEAS*, 2004.

[2] L. M. Manevitz and M. Yousef, "One-class svms for document classification," *Journal of Machine Learning Research*, vol. 2, no. Dec, pp. 139–154, 2001.

[3] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up?: Sentiment classification using machine learning techniques," in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, Association for Computational Linguistics, 2002, pp. 79–86.

[4] H. Chen, C. Schuffels, and R. Orwig, "Internet categorization and search: A self-organizing approach," *Journal of visual communication and image representation*, vol. 7, no. 1, pp. 88–102, 1996.

[5] M. K. Dalal and M. A. Zaveri, "Automatic text classification: A technical review," *International Journal of Computer Applications*, vol. 28, no. 2, pp. 37–40, 2011.

[6] A. Khan, B. Baharudin, L. H. Lee, and K. Khan, "A review of machine learning algorithms for text-documents classification," *Journal of advances in information technology*, vol. 1, no. 1, pp. 4–20, 2010.

[7] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[8] R. D. Goyal, "Knowledge based neural network for text classification," in *Granular Computing, 2007. GRC 2007. IEEE International Conference on*, IEEE, 2007, pp. 542–542.

[9] D. Isa, L. H. Lee, V. Kallimani, and R. Rajkumar, "Text document preprocessing with the bayes formula for classification using the support vector machine," *IEEE Transactions on Knowledge and Data engineering*, vol. 20, no. 9, pp. 1264–1272, 2008.

[10] P. Yuan, Y. Chen, H. Jin, and L. Huang, "Msvm-knn: Combining svm and k-nn for multi-class text classification," in *Semantic Computing and Systems, 2008. WSCS'08. IEEE International Workshop on*, IEEE, 2008, pp. 133–140.

[11] B. M. Schulze, *Automatic language identification using both n-gram and word information*, US Patent 6,167,369, Dec. 2000.

[12] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in neural information processing systems*, 2015, pp. 649–657.

[13] D. Klein, J. Smarr, H. Nguyen, and C. D. Manning, "Named entity recognition with character-level models," in *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, Association for Computational Linguistics, 2003, pp. 180–183.

[14] F. Peng, D. Schuurmans, S. Wang, and V. Keselj, "Language independent authorship attribution using character level language models," in *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics-Volume 1*, Association for Computational Linguistics, 2003, pp. 267–274.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[16] A. Ashari, I. Paryudi, and A. M. Tjoa, "Performance comparison between naıve bayes, decision tree and k-nearest neighbor in searching alternative design in an energy simulation tool," *Int. J. Adv. Comput. Sci. Appl*, vol. 4, no. 11, pp. 33–39, 2013.

[17] N. Bhatia *et al.*, "Survey of nearest neighbor techniques," *arXiv preprint arXiv:1007.0085*, 2010.

[18] H. Drucker, D. Wu, and V. N. Vapnik, "Support vector machines for spam categorization," *IEEE Transactions on Neural networks*, vol. 10, no. 5, pp. 1048–1054, 1999.

[19] A. Srivastava, E.-H. Han, V. Kumar, and V. Singh, "Parallel formulations of decision-tree classification algorithms," in *High Performance Data Mining*, Springer, 1999, pp. 237–261.