# Lab report David Bielik and Debora Beuret

# Data

The data we were given was not yet it its final form. There were more labels than tweets in total. For this reason, the methodology we followed was very simple: create pandas dataframes for both datasets (the labels and the tweets) and then merge them with the IDs. With the module pandas, the process was quick and efficient.

## Properties of the data

| | |
|---|---|
| en | 30109 |
| ja | 25487 |
| es | 8444 |
| und | 8248 |
| pt | 4578 |
| id | 4509 |
| ar | 3898 |
| ru | 2643 |

The frequency of the languages was not uniform. Almost half of the tweets were written in either English or Japanese. There were a lot of outliers with only a single occurrence in the whole dataset which created some **class imbalance**. Taking into account the distribution of the languages, we have decided to put all of the low frequency labels into one class: the 'unknown' class. More specifically, the threshold value to determine inclusion in this class is 10 (i.e. there need to be at least 10 tweets in one language).
EDIT after the whole process (13.10.2019, 20 pm) : we have realized, a little too late, that we only thought of dealing with the class imbalance on the lower end of the tail. We didn't do this for the data that was overwhelmingly present (English, Spanish, Japanese) and this might be a threat to the whole process. Indeed, it might have caused something called "mode collapse", a phenomenon in which the model predicts undiversified labels (or even a single label in the worst case) to maximize accuracy.

Another option would be to simply remove all of the low frequency labels from the training set.

After thoroughly inspecting the labels, we have found that some labels were enclosed in **unnecessary whitespace**. For example, labels 'ar ' and ' ar ' are presumably describing the same language. However, the classifier would consider these as different labels. Thus, we have simply removed all whitespace from every label.

```
train.describe()
```

| | Tweet | Label |
|---|---|---|
| **count** | 48122 | 48122 |
| **unique** | 48054 | 75 |
| **top** | :( | en |
| **freq** | 7 | 16896 |

```
dev.describe()
```

| | Tweet | Label |
|---|---|---|
| **count** | 5347 | 5347 |
| **unique** | 5346 | 43 |
| **top** | Goodmorning | en |
| **freq** | 2 | 1868 |

The final training set shows that there are there are 48122 instances in total. Out of those, only 48054 tweets are unique, which means some tweets must have appeared multiple times in the dataset. Those are presumably re-tweets, short statements or simple replies (such as ‘:(‘).

# Methodology

To choose the best performing model, the following methodology was chosen: the test set was set to fit the size and content of the file *labels-test.stv* (excluding the deleted tweets). The preprocessed *labels-train+dev.tsv* set was first shuffled and then parted: 90% for training set, 10% for development/validation set (NB: the terms “development” and “validation” may be used interchangeably in this report).

From there, we performed several **GridSearches** (cv = 4) with various parameters on the train set. The top three performers of the GridSearch would be tested with the development set. Since the model had not “seen” this data yet, a poorer performance of the best model would prove the model had overfitted the training data. It also allowed to make sure the number 1 ranking model was really the best.
Our final model would be the one that performed best on the development set. This final model would then be trained on the entire training set (90% train + 10% dev) to not lose any valuable information and increase the data. At last, we measured and reported its performance on the test set.

# Part 1 – Language identification with linear classification

## Pipelines

We considered quite a few pipelines. Obviously, **ngrams** and **tfidf** came into question and were used in our pipeline. Additionally, we tried adding the **average word length of the tweets** (see class AverageWordLengthExtractor). Initially, it had a negative impact on the performance of each classifier! The solution to this problem was to use a MinMaxScaler on the average word length values. This scaler would scale down the values to a range of [0, 1] – performing a feature scaling technique called normalization. Another side-effect of using this scaler was an obvious speed up in training times in each classifier.

We also considered **removing stop words**, but it probably wouldn't have been a good idea as they are probably important in the classifier, allowing it to identify a given language. We, for a minute, considered lemmatizing/stemming the words, but since those modules rely on already knowing what language is used, this option soon proved impossible. For this reason, we have a limited number of features.

## SGD Classifier

The parameters we tested were those described in the exercise: loss function, regularizations and number of iterations.
In a first GridSearch, we tried a few parameters (loss function, penalty, number of iterations), which resulted in this ranking.
Class imbalance, as explained earlier, was dealt with by replacing the rare labels (frequency: <10) by an 'unknown' label.

| rank_test_score | param_SGD_clf__loss | param_SGD_clf__max_iter | param_SGD_clf__penalty | mean_test_score |
|---|---|---|---|---|
| 1 | hinge | 200 | none | 0.835709 |
| 2 | hinge | 100 | none | 0.835564 |
| 3 | hinge | 100 | l2 | 0.794917 |
| 4 | hinge | 200 | l2 | 0.792216 |
| 5 | log | 100 | none | 0.789348 |
| 6 | log | 200 | none | 0.789265 |
| 7 | hinge | 200 | l1 | 0.724596 |
| 8 | hinge | 100 | l1 | 0.717219 |
| 9 | log | 200 | l1 | 0.687170 |
| 10 | log | 100 | l1 | 0.686277 |
| 11 | log | 100 | l2 | 0.673767 |
| 12 | log | 200 | l2 | 0.673289 |

We realized that the best models would consistently have a hinge loss function. The number of max_iterations would vary and both no penalty and L2 seemed to provide good results.

For this reason, we trained a new GridSearch that always had a hinge loss function. The varying parameters were the penalty (None or L2) and the maximum number of iterations (100, 150 and 200).

| | rank_test_score | param_SGD_clf__max_iter | param_SGD_clf__penalty | mean_test_score |
|---|---|---|---|---|
| **4** | 1 | 200 | none | 0.843440 |
| **2** | 2 | 150 | none | 0.842442 |
| **0** | 3 | 100 | none | 0.841819 |
| **1** | 4 | 100 | l2 | 0.798450 |
| **5** | 4 | 200 | l2 | 0.798450 |
| **3** | 6 | 150 | l2 | 0.798429 |

In this case, we saw that all the best models had no penalty. Seeing the best models, we see that the best has maximum 200 iterations, then 150 and finally 100. This made us wonder if even more than 200 would do even better, or if it was a sweet spot already. For this reason, we tried another GridSearch with a constant hinge function, no penalty. The maximum number of iterations could vary between 200 and 300. We also chose to work with one more parameter, namely the ngram range. We decided to try both (1, 2) and (1, 3) and compared the results.

| | rank_test_score | param_SGD_clf__max_iter | param_feats__ngram_tfidf__ngram__ngram_range | mean_test_score |
|---|---|---|---|---|
| **2** | 1 | 300 | (1, 2) | 0.848905 |
| **0** | 2 | 200 | (1, 2) | 0.848718 |
| **3** | 3 | 300 | (1, 3) | 0.844936 |
| **1** | 4 | 200 | (1, 3) | 0.844915 |

The results showed that the best model was having a maximum of 300 iterations and an ngram range of (1,2). It was outperforming the previous models.
In a last attempt to see if 300 was our “sweet spot” in terms of maximum iterations, we did a pipeline (no GridSearch) which had 400 iterations and performed very slightly under the winner model shown above.

**For this reason, we chose the model with the following parameters:**
**loss function: hinge**
**regularization: None**
**# of iterations: 300**
**ngram_range: (1, 2)**

**Accuracy obtained on average: 85.6%**

## Multinomial Naïve Bayes Classifier

The first parameters we tested were:

- alpha: 0.2, 0.6, 0.8, 1.0
- Fit_prior: True, False

Resulting in this:

| rank_test_score | param_features__ngram_tfidf__ngram__ngram_range | param_nb_clf__alpha | param_nb_clf__fit_prior | mean_test_score |
|---|---|---|---|---|
| 1 | (1, 2) | 0.2 | False | 0.844728 |
| 2 | (1, 4) | 0.2 | False | 0.840946 |
| 3 | (1, 2) | 0.6 | False | 0.831179 |
| 4 | (1, 2) | 0.8 | False | 0.826337 |
| 5 | (1, 4) | 0.6 | False | 0.825776 |
| 6 | (1, 2) | 1 | False | 0.821973 |
| 7 | (1, 4) | 0.8 | False | 0.820290 |
| 8 | (1, 4) | 1 | False | 0.816176 |
| 9 | (1, 2) | 0.2 | True | 0.675346 |
| 10 | (1, 4) | 0.2 | True | 0.660363 |
| 11 | (1, 2) | 0.6 | True | 0.625223 |
| 12 | (1, 2) | 0.8 | True | 0.606646 |
| 13 | (1, 4) | 0.6 | True | 0.604131 |
| 14 | (1, 2) | 1 | True | 0.590063 |
| 15 | (1, 4) | 0.8 | True | 0.582603 |
| 16 | (1, 4) | 1 | True | 0.565209 |

We then decided to try a few more models and created a new GridSearch that would always have a fit_prior set to False, since all the best performers showed that. On the other hand, there was some more play with the ngram range: (1, 2), (1, 3) and (1, 4) and the alpha was varying between 0.2, 0.4 and 0.6. We can see that the best performing model above, interestingly, is still the best (although its mean_test_score is not exactly the same).

| | rank_test_score | param_features__ngram_tfidf__ngram__ngram_range | param_nb_clf__alpha | mean_test_score |
|---|---|---|---|---|
| **0** | 1 | (1, 2) | 0.2 | 0.843938 |
| **3** | 2 | (1, 3) | 0.2 | 0.842047 |
| **6** | 3 | (1, 4) | 0.2 | 0.840073 |
| **1** | 4 | (1, 2) | 0.4 | 0.835855 |
| **4** | 5 | (1, 3) | 0.4 | 0.833195 |
| **7** | 6 | (1, 4) | 0.4 | 0.831719 |
| **2** | 7 | (1, 2) | 0.6 | 0.830286 |
| **5** | 8 | (1, 3) | 0.6 | 0.827584 |
| **8** | 9 | (1, 4) | 0.6 | 0.825735 |

Since NB is not computationally expensive, a last test was created, this time focusing on the alpha: since all the good performers had a low alpha parameter, it was legitimate to wonder if even lower would do better. For this purpose, the GridSearch was made with the alpha as only varying parameter (0.0, 0.1 or 0.2).

| | rank_test_score | param_nb_clf__alpha | mean_test_score |
|---|---|---|---|
| **1** | 1 | 0.1 | 0.870018 |
| **0** | 2 | 0 | 0.850193 |
| **2** | 3 | 0.2 | 0.846702 |

The results show that 0.1 (and even 0) perform better than 0.2. Thus, 0.1 was chosen as final alpha parameter.

**The best parameters thus were:**
**alpha = 0.1**
**fit prior = False**
**ngram_range = (1, 2)**

**Accuracy obtained on average: 87.75%**

# Part 2 – Multilayer Perceptron (MLP)

We have used the same features for MLP as for the other classifiers. Parameter-wise we have tested different hidden layer sizes, activation functions, solvers, iterations and momentum values. The initial tests were performed with cross validation via GridSearch. First thing that we noticed were the long execution times. We had to let the GridSearch run overnight in order to get some results. After some discussion we have discarded cross validation and reduced the layer size from (100,) to (5,3) and (4,3). The max iteration times were also reduced to 50. The next GridSearch was performed with a cv value of 3 resulting in this table:

| Total time (in s) | Activation | Hidden layers | Max Iter. | Momentum | Solver | Mean test score | Rank |
|---|---|---|---|---|---|---|---|
| 353.529966 | relu | (4, 3) | 50 | 0.9 | adam | 0.704647 | 1 |
| 373.916279 | tanh | (4, 3) | 50 | 0.9 | adam | 0.661007 | 2 |
| 547.581941 | relu | (5, 3) | 50 | 0.9 | adam | 0.652716 | 3 |
| 474.193237 | tanh | (5, 3) | 50 | 0.9 | adam | 0.612651 | 4 |
| 277.731456 | tanh | (4, 3) | 50 | 0.9 | sgd | 0.351128 | 5 |
| 357.605429 | tanh | (5, 3) | 50 | 0.9 | sgd | 0.351128 | 5 |
| 264.683728 | relu | (4, 3) | 50 | 0.9 | sgd | 0.351128 | 5 |
| 340.663349 | relu | (5, 3) | 50 | 0.9 | sgd | 0.351128 | 5 |

We have concluded, that the next set of tests should be done on a single hidden layer ranging from 3 to 5 hidden neurons without cross validation. As the solver Adam was the obvious choice. The momentum parameter was also left out as it has no effect when used with the Adam optimizer/solver.

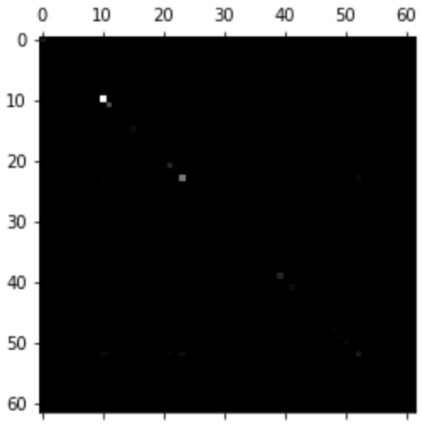
The best performance (albeit the slowest, ca 20-25min) was observed on the 5 hidden neuron MLP. This performance came with the downside of overfitting on the training set. Thus, we have played with the alpha regularization parameter which we have decided to leave at 0.01.

**The best parameters (with a reasonable execution time) were:**
**hidden_layer_sizes=(5,),**
**activation='relu',**
**solver='adam',**
**max_iter=50,**
**alpha=0.01**

**Average accuracy on the test set: 82.7%**

# Confusion matrices

All classifiers (Naïve Bayes, Stochastic Gradient Descent and MLP) resulted in visually identical confusion matrices:




The most visible squares indicate the labels for respectively English (10), Spanish (11) and Japanese (23). This is no big surprise, given that they are the most frequent labels. Almost all the other labels leave very dark squares over the diagonal, meaning, they are much less frequent in the data set. The color intensity of each square could be described as the ratio of the **T**rue **P**ositive predictions for a given label to the most frequent True Positive prediction.

Essentially, the plt.matshow function internally maps the matrix values (range: [0, TP_most_frequent]) to color intensity values (range: [0, 255]). In our case (MLP), the most frequent TP label is English with 4728 TP predictions. This would make the respective square color fully opaque (white, row 10, column 10). The second most frequent TP label is Japanese with 2222 predictions. Its color intensity is computed as follows: 255 * (TP_japanese/TP_most_frequent). TP_most_frequent is equal to TP_english. Resulting in the number 120 (row 23, column 23). We can observe these color intensity / RGB values from the picture.

# Conclusion

**GridSearchCV** was a very powerful and important tool for this exercise. It allowed us to test a wide range of parameters and compare them, testing several models in one go. It is a very efficient way to single out the best parameters and/or hint in which directions to go.
The cross validation, obviously, is very important for several reasons. It avoids overfitting by testing the model each time on a new portion of the data. It also results in a lower variance, since the model has been tested K times (K=the number of folds chosen) when the final model is set.

Overall, our **best performing model** turned out to be the Naïve Bayes Classifier. Of course, given the fact that the MLP training was bounded by time and computing power. We suspect that training it with bigger hidden layer sizes would yield even better results than NB.

# References


Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

Michelle Fullwood. *Using pipelines and Feature Union in scikit-learn*. https://michelleful.github.io/code-blog//2015/06/20/pipelines/, 2015.

Zac Stewart, http://zacstewart.com/2014/08/05/pipelines-of-featureunions-of-pipelines.html, 2014

Jeff Hale, https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02, 2019

Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2017.

Yoav Goldberg. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers, 2017

Sebastian Raschka. *Python Machine Learning*. Packt Publishing, 2015.

Delip Rao and Brian McMahan. *Natural language processing with PyTorch: build intelligent language applications using deep learning*. O'Reilly, First edition ed., 2019.