Language Detection

Mining Unstructured Data - Assignment 1

Andreja Andrejic & Danila Kokin

Introduction

The goal of this report is the note impact of different preprocessing steps and classifiers on the outcome of Language Detection. The development process consisted of two parts: preprocessing and testing different classifiers.

Baseline

The base model has no preprocessing steps and uses Naive Bayes classification method.

Character tokenization

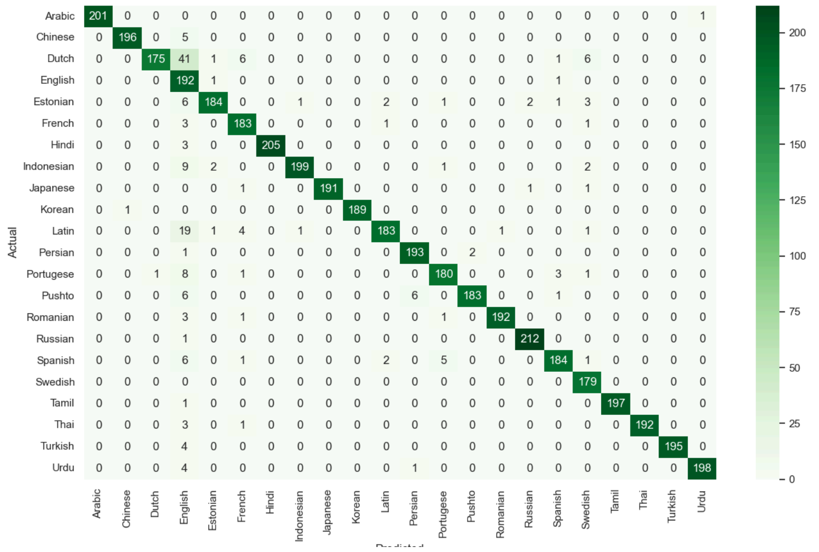


Fig. 1: Confusion matrix of predictions produced by Naive Bayes Classifier (baseline model) with 1000 vocabulary size and parameters chosen by grid search run on raw data with char tokens.

The baseline model performed well with “character” tokenization and vocabulary size 1000, having a weighted F1-score of 0.957. Looking at the confusion matrix, we see that the most misclasification belongs to Dutch and Latin languages being classified as English. This is a result of these three languages using the exact same script, with the exception of Latin language missing letters J, W and U.

Name	Size	Coverage
Raw data char-split	1000	0.9808517331929401

Model	Params	F-1 (micro)	F-1 (macro)	F-1 (weighted)
Naive Bayes	initial	0.9552272727272727	0.9578334706797794	0.9574109803676453

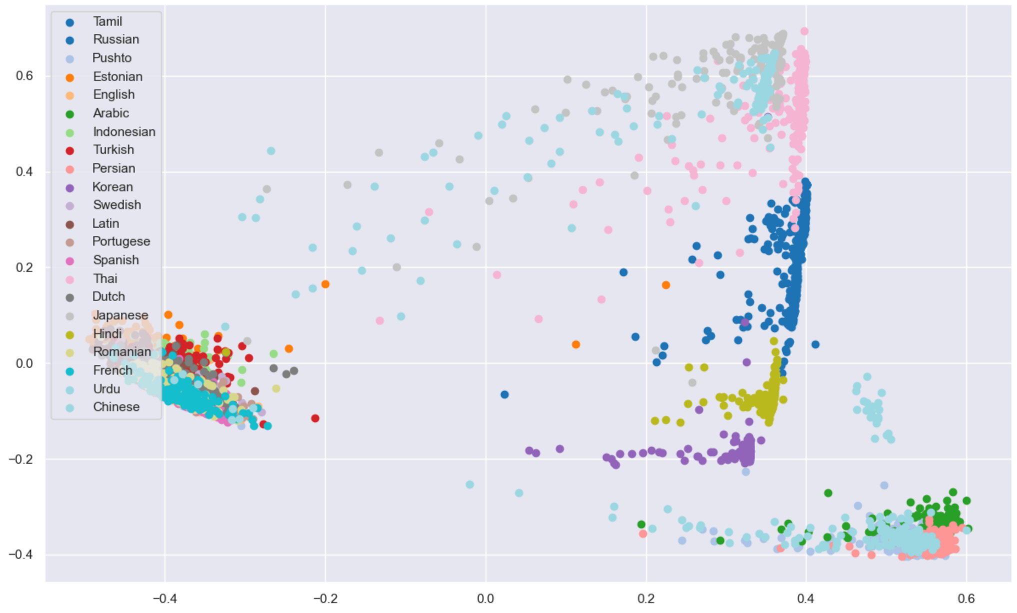


Fig. 2: Scatter plot illustrating the distribution of languages in a two-dimensional coordinate plane of principal components 1-2 (Variance explained by PCA: [0.3131436 0.13806745]) based on raw data and char tokens.

The horizontal dimension splits european languages (on the left) and the asian ones (on the right), while the vertical dimension splits the asian languages. It clearly splits Korean, Tamil, Hindi, Thai, Chinese/Japanese and languages that use Arabic script. Chinese and Japanese are partially distinguishable, because Japanese uses Chinese symbols as one of its three scripts. On the other hand, Urdu, Arabic, Persian and Pushto all use the Arabic script and are not distinguishable. Finally, this dimension does not distinguish between European languages and Hindi and Korean. By looking at the given corpus of texts, we found a lot of English text in texts marked Hindi or Korean, which explains this occurrence.

Word tokenization

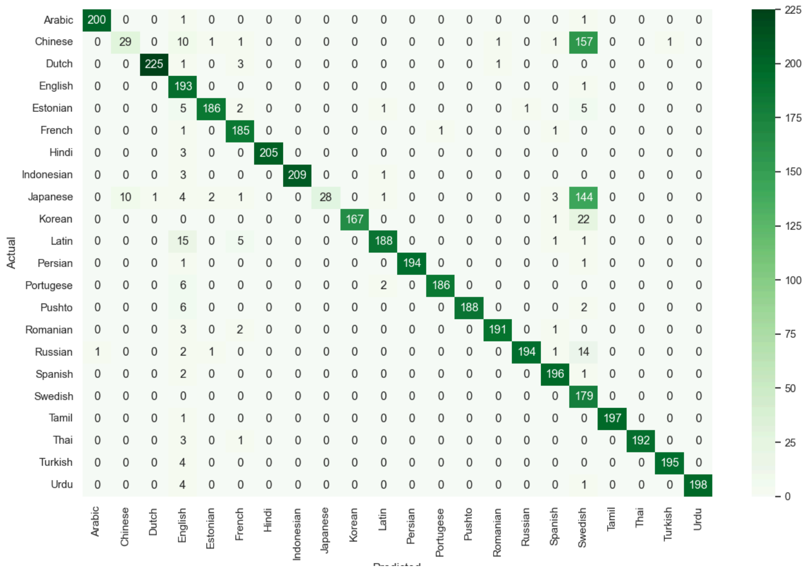


Fig. 3: Confusion matrix of predictions produced by Naive Bayes Classifier (baseline model) with 1000 vocabulary size and parameters chosen by grid search run on raw data with word tokens.

When dealing with “word” tokenization, the baseline model performs well for most langauges. Nonetheless, it has difficulties classifying Chinese and Japanese. It classifies them as Swedish and the reasoning for that is most likely due to limited vocabulary size. The CountVectorizer splits text by spaces and these two languages do not divide words with spaces. Therefore, each token is a whole sentence and so it has a much greater chance of being unique. The exact reason why it classifies them as Swedish is unknown, but probably has to do with the way CountVectorizer handles unknown tokens (new tokens arriving after the vocabulary has been filled).

Name	Size	Coverage
Raw data word-split	1000	0.25771498027437495

Model	Params	F-1 (micro)	F-1 (macro)	F-1 (weighted)
Naive Bayes	initial	0.8920454545454546	0.881225725511312	0.8845797399266582

The PCA analysis of the baseline model using “word” tokenization shown in Figure 4 does not give any sufficient insight into the data.

Preprocessing

Before the preprocessing we did a quick EDA. As the basic preprocessing steps, we lowercased all text and removed punctuation marks and excess spaces. Additionally, we generated new features related to average word length and average number of words in the sentence.

Exploratory Data Analysis

Before preprocessing and feature generation basic EDA was conducted. The statistics derived by the EDA function are presented in Table 1.

Punctuation did not give much insight on differences among languages, but other 3 statistics highlighted the differences for Chinese, Estonian, Japanese and Thai. Chinese and Japanese are highlighted by the average word length since in these languages sentences can be written without spaces at all. Since we have a problem in distinguishing Chinese and Japanese as the Japanese language uses some Chinese symbols, these features might be useful for classifiers.

Feature generation

Our feature generation function generated and added features to the vector produced by the CountVectorizer before for each row. For each sentence, these are the features added:

- Average number of spaces/words in a sentence with respect to its (character) length.
- Average word length in the sentence
- Probability of belonging to each of the languages for each sentence

The third added feature is a vector with a probability value for each language. The probabilities were assigned based on utf-8 encoding of each character. Namely, every character was classified for itself and the resulting vector has values proportional to the frequency of a symbol from each language that has occurred. Since different languages can use the same symbols, the more specific one will have been chosen, e.g. Chinese characters are classified as Chinese, even though they also appear in Japanese. The first two features are straightforward and so in total there were 24 (1 + 1 + 22) additional features. They were normalized, along with the original ones.

Three additional features were tested during experimentation: frequency of periods, commas and punctuation marks. However, they had not given additional value and were omitted from the final version.

Experiments and Results

Data coverage

When using “character” tokenization, the vocabulary coverage exceeds 98% at the size of just 1000. With “word” tokenization, we get only 26% with the size of 1000. If the size is reduced to 100, coverage drops to just 14%. And when the vocabulary is increased to 10.000 tokens, the coverage is still less than 50%, i.e. it goes up 44%. If we do the analysis on preprocessed data, the coverage increases around 1-2%.

Name	Size	Coverage
Raw data	1000	0.9808517331929401
Raw data	6816 (max)	0.9996667800941234

Table 2: Data coverage of raw and preprocessed data char-splitted with respect to vocab. size

Name	Size	Coverage
Raw data	100	0.14029876751141845
Raw data	1000	0.25771498027437495
Preprocessed data	1000	0.26805780560437625
Raw data	10000	0.43612506585579347
Preprocessed data	10000	0.4544778716949349

Table 3: Data coverage of raw and preprocessed data word-splitted with respect to vocab. size

Looking at the PCA graph in Figure 5, it is easy to distinguish two languages, namely Japanese (orange) and Chinese (brown). Their most distinguishable feature is the length of their tokens. As previously mentioned, our tokenizer splits texts by spaces, which these two languages lack. Thus when adding the average word length as a feature, we got a clear distinction between these two and other languages. Prior to adding that feature, our models were struggling mostly with this distinction and in the confusion matrix we can see that it no longer presents a problem.

Classification

As the assignment specified, we only worked on improving the system using “word” granularity. A total of three different classifiers were used, namely **Naive Bayes**, **SVM** and **XGBoost** classifiers. All three classifiers were run on a vocabulary size 1000 with “word” granularity, best parameters were chosen using grid search approach and cross-validation technique.

Naive Bayes, as the baseline model gave us the worst results illustrated in Figure 6. By the confusion matrix in Figure 7 we see that SVM gave us the second best results. Finally, the XGBoost classifier had the best results illustrated in Figure 8.

Metrics obtained during the accuracy evaluation of classifiers are shown in Table 4.

Model	Params	F-1 (micro)	F-1 (macro)	F-1 (weighted)
Naive Bayes	alpha=0.1	0.956590909090909	0.9573255732498803	0.9574392029422422
Support VM	C=10, kernel='poly'	0.9611363636363637	0.9617555362387393	0.9618324473590762

XGBoost	learning_rate=0.1, n_estimators=200	0.975909090909091	0.975909090909091	0.975909090909091

Table 4: F-1 scores of predictions produced by Naive Bayes, Support Vector Machine and XGBoost classifiers with 1000 vocabulary size and best parameters selected by Grid Search on preprocessed word-split data and generated features.

Since this model gave us the best results, we tested it on a vocabulary size 10.000 and received the following results:

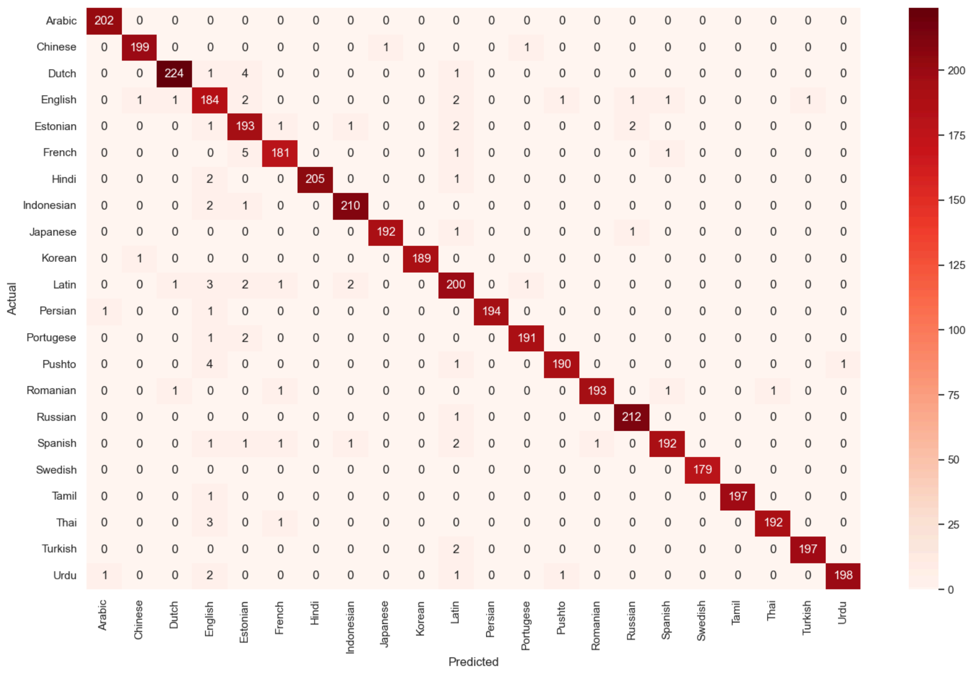


Fig. 9: Confusion matrix of prediction produced by XGBoost Classifier with 10000 vocabulary size and params of learning rate 0.1 and number of estimators 200

Model	Params	F-1 (micro)	F-1 (macro)	F-1 (weighted)
XGBoost	learning_rate=0.1, n_estimators=200	0.9804545454545455	0.9807081462727438	0.9806126560351266

Table 5: F-1 scores of predictions produced by XGBoost Classifier with 10000 vocabulary size and params of learning rate 0.1 and number of estimators 200

Results

Firstly, the data was cleaned, and after feature engineering, we selected the most optimal model, which is XGBoost, tuned its hyperparameters by setting: learning_rate=0.1, n_estimators=200 and obtained the prediction with the weighted F-score of 0.98. Taking into account that some of the rows have mixed languages and inclusions of another language, we found it to be sufficient.

Code

Preprocess

```
def preprocess(sentences, labels):
   prep_sentences = []

   for sentence in sentences:
       # preprocess text
       sentence = sentence.lower()
       sentence = remove_punctuation(sentence)
       sentence = ' '.join(sentence.split())
       prep_sentences.append(sentence)

   prep_sentences = pd.Series(prep_sentences)

   return prep_sentences, labels

def get_symbol_freq_by_family(sentence):
   symbol_families = {
       'Arabic': (0x0600, 0x06FF), 'Chinese': (0x4E00, 0x9FFF), 'Cyrillic': (0x0400, 0x04FF),
       'Turkish': (0x011E, 0x0131), 'Russian': (0x0400, 0x052F), 'English': (0x0020, 0x007F),
       'Spanish': (0x00C0, 0x024F), 'Korean': (0xAC00, 0xD7AF), 'Hindi': (0x0900, 0x097F),
       'French': (0x00C0, 0x024F), 'Thai': (0x0E00, 0x0E7F), 'Dutch': (0x0041, 0x024F),
       'Persian': (0x0600, 0x06FF), 'Pushto': (0x0620, 0x064A), 'Latin': (0x0041, 0x024F),
       'Estonian': (0x0100, 0x017F), 'Tamil': (0x0B80, 0x0BFF), 'Portuguese': (0x00C0,
0x024F),
       'Japanese': (0x3040, 0x30FF), 'Indonesian': (0x0041, 0x024F), 'Swedish': (0x0041,
0x024F),
       'Urdu': (0x0600, 0x06FF), 'Romanian': (0x0041, 0x024F)
   }

   abbreviations = {
       'Arabic': 'AR', 'Chinese': 'CN', 'Cyrillic': 'CY', 'Turkish': 'TR', 'Russian': 'RU',
       'English': 'EN', 'Spanish': 'ES', 'Korean': 'KR', 'Hindi': 'HI', 'French': 'FR',
       'Thai': 'TH', 'Dutch': 'NL', 'Persian': 'FA', 'Pushto': 'PS', 'Latin': 'LA',
       'Estonian': 'ET', 'Tamil': 'TA', 'Portuguese': 'PT', 'Japanese': 'JP', 'Indonesian':
'ID',
       'Swedish': 'SV', 'Urdu': 'UR', 'Romanian': 'RO'
   }
   total_symbols = len(sentence)
   family_freq = {abbr: 0 for abbr in abbreviations.values()}

   for char in sentence:
       for family, (start, end) in symbol_families.items():
           if start <= ord(char) <= end:
               family_freq[abbreviations[family]] += 1

   return [family_freq[abbr] / total_symbols if total_symbols > 0 else 0 for abbr in
family_freq]

def remove_punctuation(sentence):
   punctuation_chars = "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
   result = ""
   for char in sentence:
       if char not in punctuation_chars:
           result += char

   return result
```

Feature Generation

```
def add_generated_features(X_train_raw, X_test_raw, X_train, X_test):
   X_train_raw = toNumpyArray(X_train_raw)
   X_test_raw = toNumpyArray(X_test_raw)
   X_train = toNumpyArray(X_train)
   X_test = toNumpyArray(X_test)

   extrafs = get_features_for_row(X_train[0])
   train_array = np.empty([X_train_raw.shape[0], len(extrafs)])
   test_array = np.empty([X_test_raw.shape[0], len(extrafs)])

   for row in range(len(X_train)):
       train_array[row] = get_features_for_row(X_train[row])

   for row in range(len(X_test)):
       test_array[row] = get_features_for_row(X_test[row])

   return np.concatenate((X_train_raw, train_array), axis=1), np.concatenate((X_test_raw,
test_array), axis=1)

def get_features_for_row(row_data):
   features = get_symbol_freq_by_family(row_data)
   features.append(get_spaces_ratio(row_data))
   # features.append(get_commas_ratio(row_data))
   # features.append(get_periods_ratio(row_data))
   # features.append(get_special_symbols_ratio(row_data))
   features.append(get_avg_word_length(row_data))
   return np.asarray(features)

def get_spaces_ratio(sentence):
   cnt = 0.0
   for character in sentence:
       if character == ' ':
           cnt += 1.0
   return cnt / len(sentence) if len(sentence) > 0 else 0

def get_avg_word_length(sentence):
   sum = 0.0
   cnt = 0.0
   words = sentence.split(" ")
   for w in words:
       sum += len(words)
       cnt += 1.0
   return sum / cnt if cnt > 0 else 0
```

Classifiers

```
def applyNaiveBayes(X_train, y_train, X_test):
    param_grid = {
        'alpha': [0.1, 0.5, 1.0],
    }
    clf = GridSearchCV(MultinomialNB(), param_grid, cv=5)
    clf.fit(X_train, y_train)

    # Get the best estimator
    best_clf = clf.best_estimator_
    print("Naive Bayes:", best_clf)
    y_predict = best_clf.predict(X_test)

    return y_predict


def applySVM(X_train, y_train, X_test):
    param_grid = {
        'C': [0.1, 1, 10],
        'kernel': ['linear', 'rbf', 'poly'],
    }
    clf = GridSearchCV(SVC(), param_grid, cv=5)
    clf.fit(X_train, y_train)

    # Get the best estimator
    best_clf = clf.best_estimator_
    print("Support Vector Machine:", best_clf)
    y_predict = best_clf.predict(X_test)

    return y_predict


def applyXGBoost(X_train, y_train, X_test):
    # Encode labels
    label_encoder = LabelEncoder()
    y_train_encoded = label_encoder.fit_transform(y_train)
    param_grid = {
        'n_estimators': [100, 200, 300],
        'learning_rate': [0.05, 0.1, 0.2],
    }
    clf = GridSearchCV(XGBClassifier(), param_grid, cv=5)
    clf.fit(X_train, y_train_encoded)
    best_clf = clf.best_estimator_
    print("XGBoost Classifier:", best_clf)
    y_predict_encoded = best_clf.predict(X_test)

    # Decode predictions
    y_predict = label_encoder.inverse_transform(y_predict_encoded)

    return y_predict
```

Conclusion

In summary, we concluded that feature generation led by exploratory data analysis had the biggest impact on the outcome of the prediction. The baseline Naive Bayes model performed well after preprocessing. The SVM did a slightly better job, but took significantly more time to execute. Finally, the XGBoost model gave us the best prediction outcomes, as well as efficient performance. As for vocabulary size, most of the experiments were conducted on 1000 tokens and the final model was executed on 10.000 as well. While it did improve coverage, the F1-score was already high and did not improve much.

Appendix

Language	Avg Commas per Sentence	Avg Periods per Sentence	Avg Spaces per Sentence	Avg Word Length	Avg Number of Words in Sentence
Arabic	0.0	0.0	0.172980	5.873472	69.273
Chinese	0.0	0.0	0.009748	73.149249	3.196
Dutch	0.0	0.0	0.159828	6.353567	53.178
English	0.0	0.0	0.168495	6.040134	67.623
Estonian	0.0	0.0	0.128671	7.934931	38.390
French	0.0	0.0	0.167391	6.155945	63.779
Hindi	0.0	0.0	0.189360	5.315582	85.553
Indonesian	0.0	0.0	0.142475	7.123782	54.806
Japanese	0.0	0.0	0.010145	70.676142	3.131
Korean	0.0	0.0	0.230750	4.277785	62.246
Latin	0.0	0.0	0.140073	7.411036	42.191
Persian	0.0	0.0	0.193059	5.245561	77.553
Portugese	0.0	0.0	0.168707	6.008793	63.456
Pushto	0.0	0.0	0.211187	4.844383	95.446
Romanian	0.0	0.0	0.160654	6.324435	52.858
Russian	0.0	0.0	0.138021	7.384488	53.029
Spanish	0.0	0.0	0.169203	5.957845	64.452
Swedish	0.0	0.0	0.159262	6.528880	48.442
Tamil	0.0	0.0	0.110857	9.098433	68.474
Thai	0.0	0.0	0.051565	20.843826	68.474
Turkish	0.0	0.0	0.133885	7.571763	68.474
Urdu	0.0	0.0	0.205556	4.891565	68.474

Table 1. EDA statistics of the raw sentences with outstanding values highlighted.

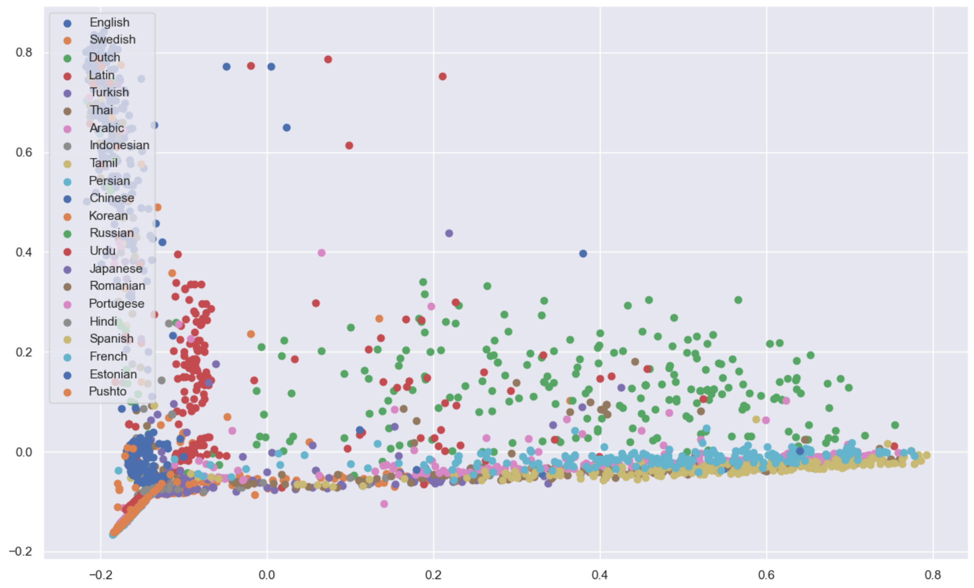


Fig. 4: Scatter plot illustrating the distribution of languages in a two-dimensional coordinate plane of principal components 1-2 (Variance explained by PCA: [0.07878438 0.03638188]) based on raw data and word tokens.

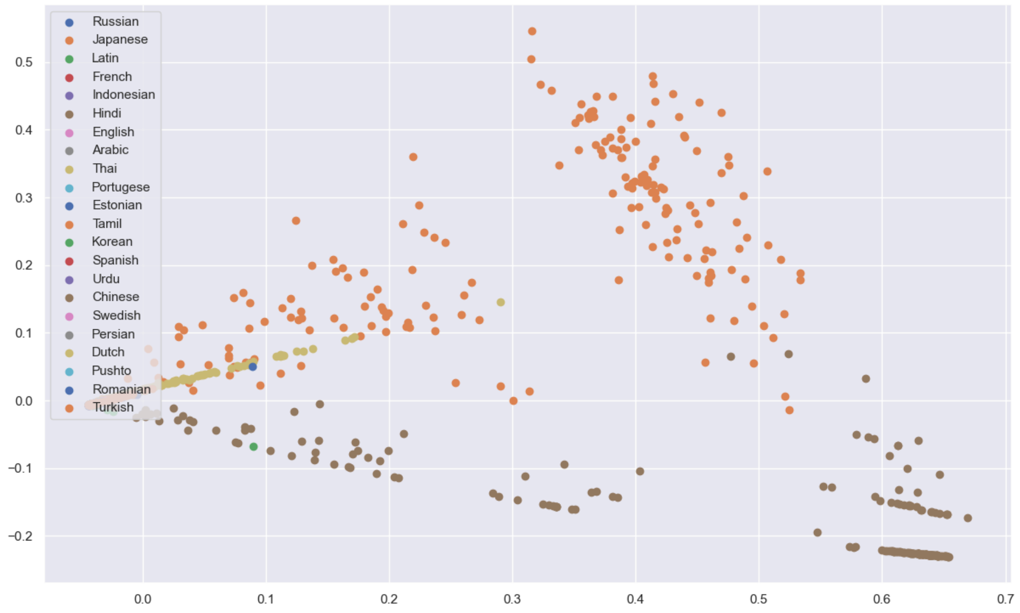


Fig. 5: Scatter plot illustrating the distribution of languages in a two-dimensional coordinate plane of principal components 1-2 based on preprocessed data split by words combined with generated features.

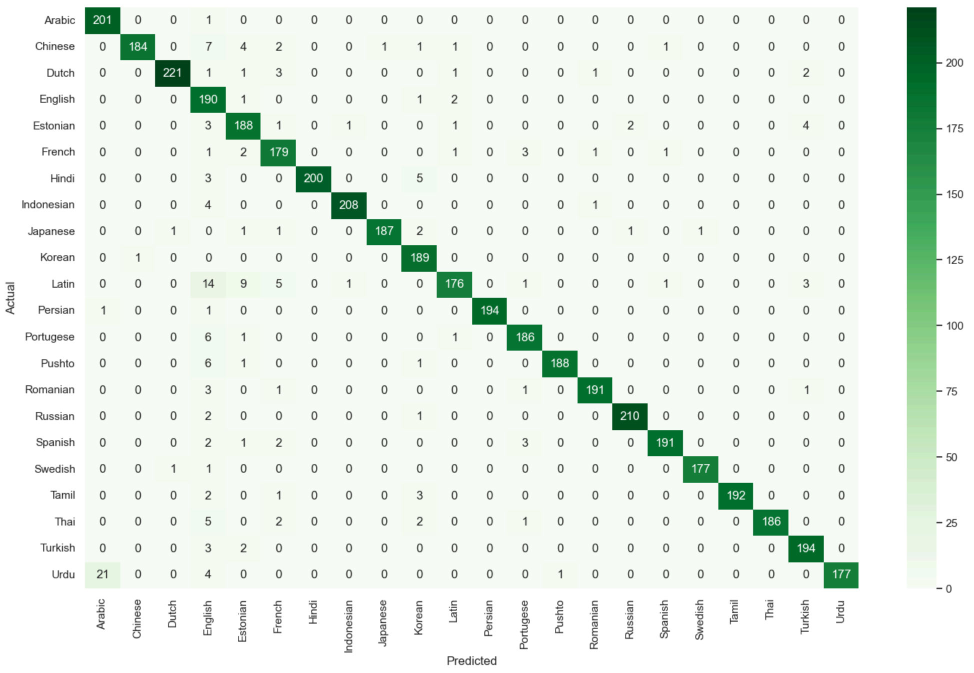


Fig. 6: Confusion matrix of prediction produced by Naive Bayes Classifier with 1000 vocabulary size and parameters chosen by grid search on preprocessed word-split data and generated features.

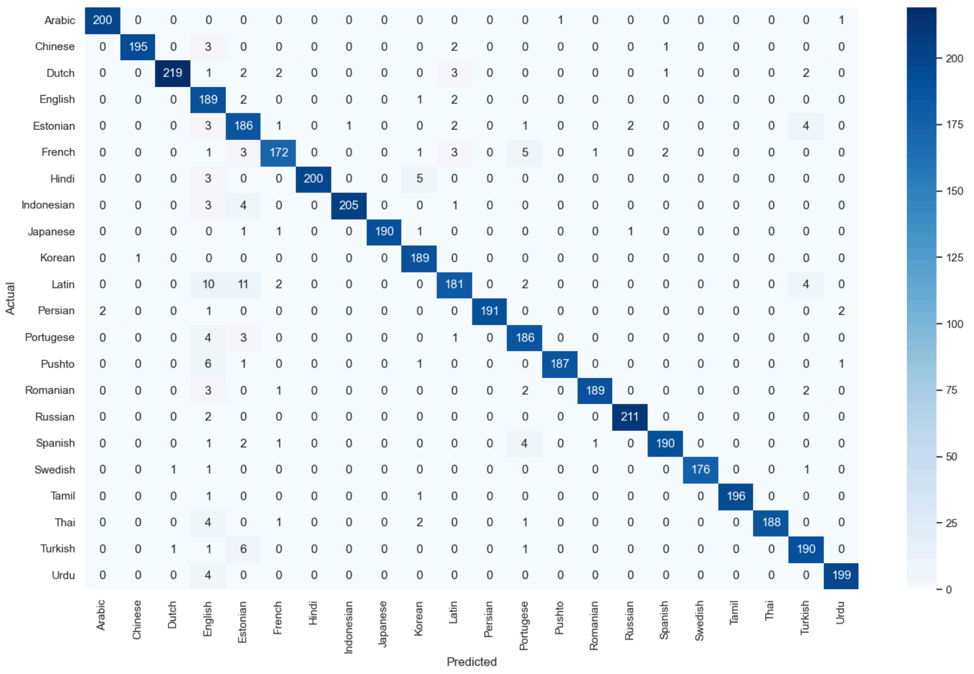
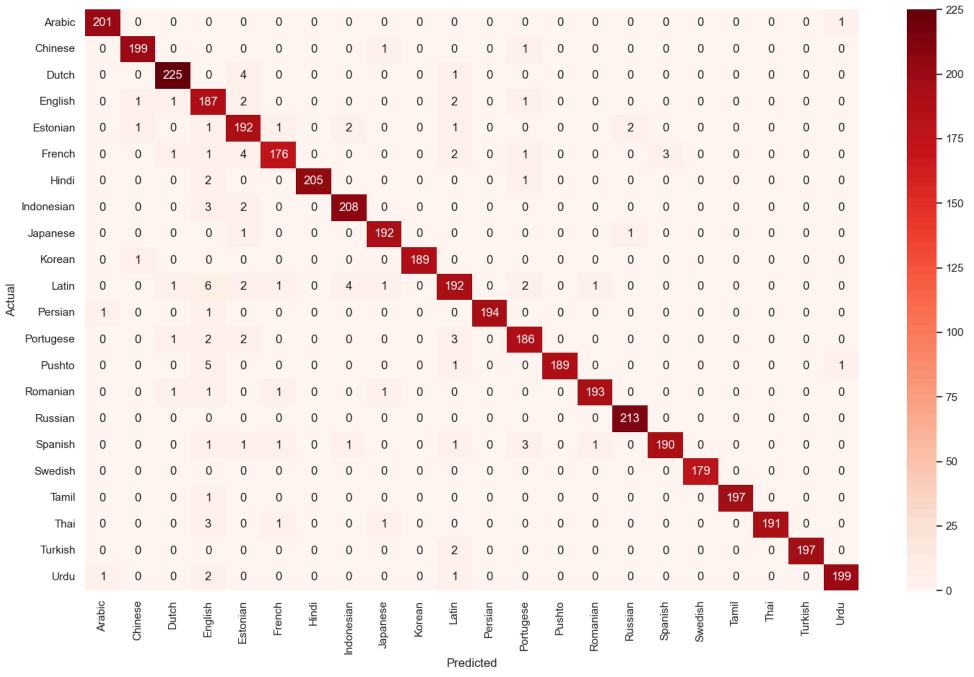


Fig. 7: Confusion matrix of prediction produced by Support Vector Machine Classifier with 1000 vocabulary size and parameters chosen by grid search on preprocessed word-split data and generated features.



	Arabic	Chinese	Dutch	English	Estonian	French	Hindi	Indonesian	Japanese	Korean	Latin	Persian	Portugese	Pushto	Romanian	Russian	Spanish	Swedish	Tamil	Thai	Turkish	Urdu
Arabic	201	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Chinese	0	199	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
Dutch	0	0	225	0	4	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
English	0	1	1	187	2	0	0	0	0	0	2	0	1	0	0	0	0	0	0	0	0	0
Estonian	0	1	0	1	192	1	0	2	0	0	1	0	0	0	0	2	0	0	0	0	0	0
French	0	0	1	1	4	176	0	0	0	0	2	0	1	0	0	0	3	0	0	0	0	0
Hindi	0	0	0	2	0	0	205	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Indonesian	0	0	0	3	2	0	0	208	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Japanese	0	0	0	0	1	0	0	0	192	0	0	0	0	0	0	1	0	0	0	0	0	0
Korean	0	1	0	0	0	0	0	0	0	189	0	0	0	0	0	0	0	0	0	0	0	0
Latin	0	0	1	6	2	1	0	4	1	0	192	0	2	0	1	0	0	0	0	0	0	0
Persian	1	0	0	1	0	0	0	0	0	0	0	194	0	0	0	0	0	0	0	0	0	0
Portugese	0	0	1	2	2	0	0	0	0	0	3	0	186	0	0	0	0	0	0	0	0	0
Pushto	0	0	0	5	0	0	0	0	0	0	1	0	0	189	0	0	0	0	0	0	0	1
Romanian	0	0	1	1	0	1	0	0	1	0	0	0	0	0	193	0	0	0	0	0	0	0
Russian	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	213	0	0	0	0	0	0
Spanish	0	0	0	1	1	1	0	1	0	0	1	0	3	0	1	0	190	0	0	0	0	0
Swedish	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	179	0	0	0	0
Tamil	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	197	0	0	0
Thai	0	0	0	3	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	191	0	0
Turkish	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	197	0
Urdu	1	0	0	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	199

Fig. 8: Confusion matrix of prediction produced by XGBoost Classifier with 1000 vocabulary size and parameters chosen by grid search on preprocessed word-split data and generated features.