Alien language classification

Tiganus Alexandru-Daniel, 232

Objective at glance

The objective is to train a model that can accurately predict what language a text sample belongs to, from the 3 discovered alien languages. Basically, this is a task of language processing, which nowadays is used in many technologies we use daily such as but not limited to: spam detection, voice recognition, language recognition and so on.

The code

The program is using functions for all operations for the code to be easily readeable. It can be expanded upon easily as the comments explain pretty decently what's going on. The get_data function fetches the data from the files given. The get_features function gets the features of each sample, the predict function predicts the test features assigning them labels, and the write function write the results in a file. The results function prints the confusion matrix and other statistics for the validation features. The program can be changed from generating a result file for the test data to generating statistics for the validation data by decommenting a line:

```
if __name__ == "__main__":
    start_time = time.time()

    # Get results for test data
    predict_test_data()

    # Get statistics for validation data
    #predict_validation_data()

    end_time = time.time()
    print(f"Execution time: {end_time - start_time} seconds")
```

The predict test data function looks like this, including some of the functions mentioned above:

```
def predict_test_data():
    # Get data
    train_data, validation_data, test_data, train_index, validation_index, test_index, train_labels, validation_labels = get_data(merge = True)

    # Get the features
    train_features, test_features = get_features(train_data, test_data)

    # Train and predict
    prediction = predict(train_features, test_features)

    # Write file
    write(prediction, test_index)
```

The predict validation data function looks like this. They're very alike and self-explanatory:

```
def predict_validation_data():
    # Get data
    train_data, validation_data, test_data, train_index, validation_index, test_index, train_labels, validation_labels = get_data(merge = False)

    # Get the features validation included
    train_features, validation_features = get_features(train_data, test_data)

    # If we want to predict the validation features
    prediction = predict(train_features, validation_features)

    # Print confusion matrix and other data for validation data
    results(prediction, validation_data)
```

Getting the data

The data given in 3 categories, train data, which composed of train_samples.txt and train_labels.txt. It contained the training sentences that are used to train the model and their corresponding labels.

Validation data is given in the validation_samples.txt and validation_labels.txt files. It can be used to help training the model by using it basically encomposed in the train data. Also, it can be used for checking results such as confusion matrix and accuracy of different models quickly, by using only the samples like test samples.

Test data is given in the test_samples.txt file and contains the data that needs labeling.

Each data sample and label is given an index, but we'll only need the index for the test samples in order to generate the final result csv.

The data is read using the genfromtxt numpy function, using a delimiter to create a separation between the indexes and labels/text. The

mbcs encoder gives the best results with the dataset, a bit better than utf-8. Also we'll need the comments = None parameter because otherwise '#' will be taken as a start comment character in the input file and it will not read the full sentences properly.

```
def get_data(merge = False):
    # Read data
    print("Getting data...")
    data_time = time.time()

    train_data = np.genfromtxt(path + '\data\\train_samples.txt', dtype='str', encoding='mbcs', delimiter = '    ', comments = None)
    train_labels = np.genfromtxt(path + '\data\\train_labels.txt', delimiter = '    ', dtype='int')
    validation_data = np.genfromtxt(path + '\data\\validation_samples.txt', dtype='str', encoding='mbcs', delimiter = ' ', comments = None)
    validation_labels = np.genfromtxt(path + '\data\\validation_labels.txt', delimiter = '  ', dtype='int')
    test_data = np.loadtxt(path + '\data\\test_samples.txt', dtype='str', encoding='mbcs', delimiter = '    ', comments = None)
```

The data than is put in its appropriate arrays, using the delimiter to make the distinction between the indexes and the data/labels.

```
# Initial indexes
test_index = [i[0] for i in test_data]
train_index = [i[0] for i in train_data]
validation_index = [i[0] for i in validation_data]

# Data
train_data = [i[1] for i in train_data]
test_data = [i[1] for i in test_data]
validation_data = [i[1] for i in validation_data]

# Get labels
train_labels = [i[1] for i in train_labels]
validation_labels = [i[1] for i in validation_labels]
```

Then, if we want, we'll merge the validation data with the train data to obtain a bigger train data pool (15000 vs 10000).

```
# Take validation data into consideration
if merge:
    train_data = np.append(train_data, validation_data)
    train_labels = np.append(train_labels, validation_labels)

print(f"Getting data complete, it took: {time.time() - data_time} seconds.")

return train_data, validation_data, test_data, train_index, validation_index, test_index, train_labels, validation_labels
```

This whole operation takes about 1 second, which given the length of the dataset is decent.

```
Getting data...
Getting data complete, it took: 0.9517569541931152 seconds.
```

Generating features

Having the data saved we'll need to extract features from it to feed our model. This is done using the Bag Of Words method. How it works is it creates a vocabulary of words or group of characters from some initial data, and assigns an id for every word. Then, we represent each data sample as a vector of length equal to the vocabulary's length, where every member of the vector represents the number of times the word with the id equal to the vector's index appears in the sample. An example:

	about	bird	heard	is	the	word	you
About the bird, the bird, bird bird bird	1	5	0	0	2	0	0
You heard about the bird	1	1	1	0	1	0	1
The bird is the word	0	1	0	1	2	1	0

Image taken from [openclassrooms.com]

CountVectorizer was used for this procedure. It was initialized with parameters found using grid search, basically putting random parameters and seeing if the results improve or not using the validation data until a pattern was found and the parameters good for our dataset were found.

```
def get_features(train_data, test_data):
    # BagOfWords with countvectorizer
    print("Getting features...")
    features_time = time.time()

    # Initialize countvectorizer
    vectorizer = CountVectorizer(ngram_range=(3, 7),
                                 analyzer="char",
                                 strip_accents="unicode",
                                 binary=True,
                                 encoding="mbcs",
                                 max_features=1220000)
```

The analyzer, strip_accents, binary, max_features and ngram_range parameters were found and optimized using grid search, while the encoding one was set to fit the genfromtxt encoder. These parameters basically state that the vocabulary should be made from group of characters ranging from 3 to 7 characters, with their accents stripped. Binary means that the words are not counted, but marked as boolean values meaning only their whether a word is present or not is marked, not the number of occurences. The max_features parameters make the vectorizer take into consideration only the top 1220000 number of features taken from the samples.

Afterwards, the vocabulary is created using the fit method, and we get the features for the data provided using the transform method.

```
# Create vocabulary
vectorizer.fit(train_data)

# Get features
train_features = vectorizer.transform(train_data)
test_features = vectorizer.transform(test_data)

print(f"Getting features complete, it took: {time.time() - features_time} seconds.\n")

return train_features, test_features
```

This whole process takes about 40 seconds.

```
Getting features...
Getting features complete, it took: 42.3139328956604 seconds.
```

Generating a prediction

For generating a prediction the sklearn library was used. A combination of 3 algorithms which performed well in the tests were used. The first one Naïve Bayes, using its Multinomial classificator. This is a simple classificator, based on Bayes' probability theorem. The features are independent from each other. This classificator achieved the best results, and is generally good for language processing. It is also the fastest by far. It was used with it an alpha = 0.1 parameter found using grid search. It is

used for Laplacian Smoothing, basically adding the alpha to both the numerator and the denominator of the Bayes' equation fraction.

The second algorithm used was Random Forest, which again is an algorithm that performs well in language processing. It is much slower than Naïve Bayes. It generates a bunch of decision trees, and the final decision is based on the deicision of the majority of the trees.

The third algorithm used was Multi Layer Percepton. This is basically a grid of perceptons, including an input layer, a few (in our case 10, found using grid search) hidden layers and an output layer of perceptons. Each layer is fed the results of the last, and using an activation function each percepton gives a prediction. It is much slower than Random Forest.

The difference between the 2 methods used is here, one using a combination of 3 algorithms, while the other only using the Naïve Bayes classifier.

For all the classifiers the fit function trains the model, while the predict function predicts the test samples.

```
def predict(train_features, test_features, train_labels):
    # Initializing the classifiers
    print("Predicting...")
    prediction_time = time.time()

    nb = MultinomialNB(alpha=0.1)
    rf = RandomForestClassifier()
    mlp = MLPClassifier(hidden_layer_sizes=10)

    # Predict using naive bayes
    nb_time = time.time()
    nb.fit(train_features, train_labels)
    predictionNB = nb.predict(test_features)
    print(f"Predicting using Naive Bayes took: {time.time() - nb_time} seconds.")

    # Predict using random forest
    rf_time = time.time()
    rf.fit(train_features, train_labels)
    predictionRF = rf.predict(test_features)
    print(f"Predicting using Random Forest took: {time.time() - rf_time} seconds.")

    # Predict using multi layer percepton
    mlp_time = time.time()
    mlp.fit(train_features, train_labels)
    predictionMLP = mlp.predict(test_features)
    print(f"Predicting using Multi-Layer Percepton took: {time.time() - mlp_time} seconds.")
```

Having the predictions of all the classifiers, we merge them using the majority rule. The final prediction is the prediction of the majority of the classifiers. If all of them predicted a different label, we trust the Naïve Bayes classifier the most because it performed best in tests, and go with its prediction.

```
# Merge the predictions of all the algorithms
prediction = []

for i in range(len(predictionNB)):
    if predictionNB[i] == predictionRF[i] or predictionNB[i] == predictionMLP[i]:
        prediction.append(predictionNB[i])
    elif predictionRF[i] == predictionMLP[i]:
        prediction.append(predictionMLP[i])
    else:
        prediction.append(predictionNB[i])

print(f"Predicting took: {time.time() - prediction_time} seconds.\n")

return prediction
```

This part of the program takes the most time, taking about 12 minutes and 40 seconds to complete. Much of this time is spent classifying using the MLPClassifier, taking about 11 minutes to complete. The Random Forest classifier took about 1 minute and 40 seconds to complete its training and prediction, while Naïve Bayes took ¼ of a second.

```
Predicting...
Predicting using Naive Bayes took: 0.3173713684082031 seconds.
Predicting using Random Forest took: 100.77443242073059 seconds.
Predicting using Multi-Layer Percepton took: 664.4945001602173 seconds.
Predicting took: 765.5892946720123 seconds.
```

Writing in file

Having the results, now perhaps the easiest part is writing them in a csv file for review. The write funciton does exactly that.

```python
def write(prediction, test_index):
    # Writing in file
    print("Writing in file...")
    write_time = time.time()

    fout = open(path + "\\naiveBayesOutput.csv", "w")
    fout.write("id,label\n")
    for i in range(len(prediction)):
        fout.write(f"{test_index[i]},{prediction[i]}\n")

    print(f"Writing in file complete, it took: {time.time() - write_time} seconds.\n")
```

Writing takes an insignificant amount of time:

```
Writing in file...
Writing in file complete, it took: 0.008011341094970703 seconds.
```

Results

The results are found using the results function, on the validation labels, without training with the validation labels of course. It returns the number of good guesses, the accuracy and the confusion matrix.

```python
def results(prediction, validation_labels):
    # Create confusion matrix and keep count of good guesses
    print("Generating results...")
    results_time = time.time()

    good = 0
    mat = np.zeros((3, 3))
    for i in range(len(validation_labels)):
        if validation_labels[i] == prediction[i]:
            good += 1
            mat[prediction[i] - 1][prediction[i] - 1] += 1
        else:
            mat[validation_labels[i] - 1][prediction[i] - 1] += 1

    # Print the data
    all = len(validation_labels)
    print(f"Good guesses: {str(good)} / {str(all)}")
    print(f"Accuracy: {str(good / all)}")
    print("Confusion matrix:")
    print(mat)
    print(f"Getting results complete, it took: {time.time() - results_time} seconds.\n")
```

Parsing the results also takes an insignifiant amount of time:

```
Generating results...
Good guesses: 3863 / 5000
Accuracy: 0.7726
Confusion matrix:
[[1688.  187.  125.]
 [ 356. 1042.  102.]
 [ 317.   50. 1133.]]
Getting results complete, it took: 0.0039618015289306640 seconds.
```

So this method obtained an accuracy of 0.7726 on the validation data, a bit higher than the one obtained on Kaggle, 0.77416. Using the confusion matrix we can determine that this method confuses the 2 and 3 alien languages with the first one a lot.

Comparison with other method

The other method that has been used is using just Naïve Bayes, without MLP and RF. On Kaggle, this method obtained a score of 0.76735, which compared to the 3 algorithms method that obtained 0.77416, is a little bit lower, more precisely 0.00681. While the improvement is small, it is crucial, as it accounts for 10 places in the leaderboard, so a lot.

However, while the results are somewhat better, the time efficiency is definitely not. The time it takes the 3 algorithm method to complete is about 800 seconds, or 13 minutes and 20 seconds, and the time it takes the Naïve Bayes method to complete is about 45 seconds. So that's 18 times faster than the algorithm used, and it gives only slightly worse results.

```
Getting data...
Getting data complete, it took: 0.9534504413604736 seconds.

Getting features...
Getting features complete, it took: 42.3139328956604 seconds.

Predicting...
Predicting using Naive Bayes took: 0.3173713684082031 seconds.
Predicting using Random Forest took: 100.77443242073059 seconds.
Predicting using Multi-Layer Percepton took: 664.4945001602173 seconds.
Predicting took: 765.5892946720123 seconds.
Writing in file...
Writing in file complete, it took: 0.004985809326171875 seconds.

Execution time: 808.9905605316162 seconds
```

```
Getting data...
Getting data complete, it took: 0.9199490547180176 seconds.

Getting features...
Getting features complete, it took: 44.55391025543213 seconds.

Predicting...
Predicting using Naive Bayes took: 0.31972575187683105 seconds.
Predicting took: 0.32018160820007324 seconds.
Writing in file...
Writing in file complete, it took: 0.004958391189575195 seconds.

Execution time: 45.94765663146973 seconds
```

When it comes to the confusion matrix and results on validation data, the results are about the same as the results obtained on Kaggle. The 3 algorithm method obtained an accuracy of 0.7726, while the Naïve Bayes one obtained an accuracy of 0.7664. Looking at the confusion matrix, it can be noticed that Naïve Bayes is slightly better at detecting correctly the first and second languages, while the other method is better at detecting correctly the first language. However, the 3 algorithm method predicts more samples to be the first language incorrectly as well, so we can notice a bias in that sense.

```
Generating results...
Good guesses: 3863 / 5000
Accuracy: 0.7726
Confusion matrix:
[[1688.  187.  125.]
 [ 356. 1042.  102.]
 [ 317.   50. 1133.]]
Getting results complete, it took: 0.0039618015289306640 seconds.
```

3 algorithm method results

```
Generating results...
Good guesses: 3832 / 5000
Accuracy: 0.7664
Confusion matrix:
[[1585.  253.  162.]
 [ 311. 1071.  118.]
 [ 258.   66. 1176.]]
Getting results complete, it took: 0.0051057338714599610 seconds.
```

Naïve Bayes method results

Hyperparameter tuning

As mentioned previously, all the parameters used for tuning were found by trial and error. Both models use the same CountVectorizer parameters for generating the features, which is what I found influenced the results the most. All the times are on the getting features method. The times are only for the Naïve Bayes method for the CountVectorizer method. They illustrate the fluctuation well.

The influence of the “binary” parameter on time is insignificant, but it increases accuracy slightly.

binary	False	True
Accuracy	0.7576	0.7664
Time	29s	30s

The influence of the “ngram_range” parameter on time is very significant. Looking at what it does, it is evident why that happens, because with a lower range, the vocabulary is smaller in size, because it takes less ngrams into consideration. For example, the range (1, 1) takes into consideration only letters, while (1, 2) takes both letters and concatenation of 2 letters. The accuracy is best while using the (3, 7) ngram, it works best on our data.

ngram_range	(1, 1)	(1, 4)	(3, 9)	(1, 7)	(3, 7)
Accuracy	0.48	0.7368	0.7586	0.7660	0.7664
Time	2s	13s	50s	34s	30s

The influence of the “max_features” parameter on time is not really that signficant, only dropping for lower values, wheras the impact on accuracy is pretty decent. Lower values obtain lower accuracy scores, while higher ones obtain better scores. The sweet spot is in the 1220000s though.

max_features	None	1000	100000	1220000	2000000
Accuracy	0.7624	0.5064	0.7038	0.7664	0.7654
Time	30s	25s	28s	30s	30s

When it comes to specific tuning, the Naïve Bayes classifier was tuned only using the alpha parameter. The time is from the execution of the predict method. It does not impact time significantly, but it impacts accuracy slightly.

alpha	1	0.5	0.1	0
Accuracy	0.7508	0.7594	0.7664	0.7118
Time	0.3s	0.3s	0.3s	0.3s

The tuning made on other classifiers in the 3 algorithm method was on the “hidden_layers” parameter of the Multi-Layer Percepton Classifier. The time was recorded and accuracy were recorded only for the MLPClassifier on its own. The accuracy seems to get better as more hidden_layers are added, but it also gets progressively more time consuming, because there are more perceptons that need to predict.

hidden_layers	100	10	1
Accuracy	?	0.7546	0.5892
Time	>2h	536s	227s

The importance of hyperparameter tuning and impact is detailed in the below table. Although it takes a longer time with the default parameters, it can predict significantly better than without them.

Hypertuning?	No	Yes
Accuracy	0.7030	0.7726
Time	250s	808s

Conclusions

In conclusion, the models described above have obtained an accuracy of about 75% on the test data, which is a lot better than the baseline of 18%. Given a text in the first language, the 3 algorithms method can predict it accurately about 85% of the time, which is great. It can predict the third langauge accurately 75% of the time, but can only predict accuretely about 69.5% of the second language. So the method works best on the first language and siginifcantly worse on the second one.

If time is a concern, the Naïve Bayes method is better because it gives a result only about 0.6% less accurate, but 18 times faster. So probably this method is to be used if need be. However, if time is absolutely not a concern, the 3 algorithms method is slightly better.