Evaluation of Hierarchical Classification

Ameet Deshpande

May 2017

1 Abstract

Two approaches for hierarchical classification and two classifiers are evaluated. First approach(called **Local Classifier Per Parent**) creates one classifier per parent in the hierarchy tree, while the second(called **Global Classifier**) treats all the leaves as separate classes and does a multi-class classification.
The two classifiers that are being used are Multinomial Naive Bayes and Linear Support Vector machine, which have both been found to do well in document classification.
The four models will be evaluated on the following metrics.

- Accuracy of Classification
- F-Score(Macro-averaged)
- Prediction Execution Time
- Training Execution Time

It is worth noting that the machine used is a windows, 4 *GB RAM* machine with an intel $i5$ processor. The dataset that is being used is the standard 20 Newsgroups Dataset. The training set consists of 10721 samples and the test set consists of 7137 samples. The code is written using python and sklearn library.

2 Local Classifier Per Parent

2.1 Multinomial Naive Bayes

- Accuracy of Classification : 67.78%
- F-Score : 0.654
- Prediction Execution Time : 22 *seconds*
- Training Execution Time : 17 *seconds*

2.2 Support Vector Machine

- Accuracy of Classification : 82.24%
- F-Score : 0.81
- Prediction Execution Time : 27 *seconds*
- Training Execution Time : 21 *seconds*

3 Global Classifier

3.1 Multinomial Naive Bayes

- Accuracy of Classification : 80.83%
- F-Score : 0.7927
- Prediction Execution Time : 5 *seconds*
- Training Execution Time : 15 *seconds*

3.2 Support Vector Machine

- Accuracy of Classification : 84.04%
- F-Score : 0.829
- Prediction Execution Time : 5 *seconds*
- Training Execution Time : 24 *seconds*

4 Analysis

In local classifier approach, I found it challenging to vectorize the prediction part of the code, as different classifiers will have to be used for different test examples. That apart, *Support Vector Machines* seem to be doing better than *Naive Bayes* approach. Local classifier has significantly larger prediction time without vectorization. One more thing to notice is that, there were not many leaves in this dataset. Thus, the global classifier approach worked here. If the number of leaves increase, the global classifier approach will take a bigger hit than the other approach.

5 Code

5.1 Driver Code

```
"""
This model implements the Naive Bayes classifier with
one classifier per parent node.
The hierarchy is assumed to be given in a file in the
format described below. Parent Node Child Node
The first node is assumed to represent the root
The labels are assumed to not have any space in
their names.

20Newsgroup Dataset is being used

It is assumed that the document can only be part of
    ↪ leaves

Author : Ameet S Deshpande
"""
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import
    ↪ CountVectorizer
from sklearn.feature_extraction.text import
    ↪ TfidfTransformer
from sklearn.pipeline import Pipeline
from sklearn import metrics
from sklearn.datasets.twenty_newsgroups import
    ↪ fetch_20newsgroups
from global_variables import stop, topic_mapping,
    ↪ inverse_mapping, leaf_to_topic,
    ↪ inverse_leaf_to_topic, cats
from functions import build_hierarchy, train_classifiers,
    ↪  build_classifier_map, predict_class
import time

start = time.clock()

train_dataset = fetch_20newsgroups(subset='train',
    ↪ categories=cats, shuffle=True, random_state=42)

# Adjacency list represents the hieararchy tree
# node_int_map maps the node label to the adjacency list
    ↪ index
```

```
# node_int_inverse_map represents the inverse of
    node_int_map
# parent_nos represents the number of nodes which are not
    leaves
[adjacency_list, node_int_map, node_int_inverse_map,
    parent_nos] = build_hierarchy()
start_time = time.process_time()
count_vectorizer = CountVectorizer(stop_words=stop,
    ngram_range=(1, 2))
tfidf_transformer = TfidfTransformer()
features = count_vectorizer.fit_transform(train_dataset.
    data)
features = tfidf_transformer.fit_transform(features)
print("-----Built Features-----")

classifier_map = build_classifier_map(adjacency_list)
# Construct one clissifier for each parent node
classifiers = [SGDClassifier(alpha=1e-3, random_state=42)
    for i in range(parent_nos)]
print("-----Created Classifiers-----")
# 0 represents the root

garbage = train_classifiers(classifiers, adjacency_list,
    0, features, np.array(train_dataset.target),
    node_int_inverse_map, leaf_to_topic, classifier_map
    )
print("-----Training Done-----")
print(time.process_time()-start_time)

test_dataset = fetch_20newsgroups(subset='test',
    categories=cats)
actual_answers = test_dataset.target
start = time.process_time()
documents = tfidf_transformer.transform(count_vectorizer.
    transform(test_dataset.data))
predictions = predict_class(documents, classifiers,
    classifier_map, leaf_to_topic, node_int_inverse_map
    , count_vectorizer, tfidf_transformer)
print(time.process_time()-start_time)
#predictions = classifiers[0].predict(documents)

f = open("predictions.txt","w")
for i in range(len(predictions)) :
        f.write("Actual Answer : "+str(actual_answers[i])
            + "Prediction : "+str(predictions[i])+"\n"
            )
```

```
print("———Testing Done———")

print("The Accuracy is : "+str(metrics.accuracy_score(
    ↪ actual_answers, predictions, normalize=True)*100))
print("The F-Score is : "+str(metrics.f1_score(
    ↪ actual_answers, predictions, average='macro')))
print("Total number of articles in the training set is : 
    ↪ "+str(len(train_dataset.target)))
print("Total number of articles in the test set is : "+
    ↪ str(len(predictions)))

print("Time Elapsed : "+str(time.clock()-start))
```

5.2 Functions

```
from global_variables import stop, topic_mapping,
    ↪ inverse_mapping, leaf_to_topic,
    ↪ inverse_leaf_to_topic
import numpy as np

# Reads the hierarchy file and builds adjacency list
def build_hierarchy():
        # Read the lines from the file
        file_pointer = open("hierarchical_structure.txt",
            ↪  "r")
        edges = file_pointer.readlines()
        file_pointer.close()

        # Variables to return
        node_int_map = {}                       # Maps
            ↪ the label of the node to an integer label
            ↪ which represents the node in the list
        node_int_inverse_map = {}       # Represents the
            ↪ inverse mapping of the previous dictionary
        adjacency_list = []             # Represents the
            ↪ tree that has been given as the input
        counter = 0
        parent_nos = 0

        # Go through all the parent-child relationship
        for edge in edges:
                edge = edge.strip('\n').split()         #
                    ↪  edge now contains a list with
                    ↪ parent, child as elements
```

```
                for i in range(2) :
                        if int(edge[i]) not in
                           ↪ node_int_map.keys() :      #
                           ↪ If a integer has not been
                           ↪ assigned to the node yet
                                node_int_map[int(edge[i])
                                   ↪ ] = counter
                                node_int_inverse_map[
                                   ↪ counter] = int(edge
                                   ↪ [i])
                                counter += 1
                                adjacency_list.append([])
                                   ↪         # Append an
                                   ↪ empty list to the
                                   ↪ adjacency list
                adjacency_list[node_int_map[int(edge[0])
                   ↪ ]].append(node_int_map[int(edge[1])
                   ↪ ])

        for node in adjacency_list :
                if node != [] :
                        parent_nos += 1

        # Return the list of the three items
        return [adjacency_list, node_int_map,
           ↪ node_int_inverse_map, parent_nos]

def train_classifiers(classifiers, adjacency_list, node,
   ↪ features, train_dataset_target,
   ↪ node_int_inverse_map, leaf_to_topic, classifier_map
   ↪ ) :
        if not adjacency_list[node] :
                documents = node
                documents = leaf_to_topic[
                   ↪ node_int_inverse_map[node]]
                boolean_array = (train_dataset_target ==
                   ↪ documents)                #
                   ↪ Vectorizing the code
                return boolean_array
        else :
                boolean_array = np.zeros(shape=len(
                   ↪ train_dataset_target), dtype=bool)
                local_target = np.ones(shape=len(
                   ↪ train_dataset_target), dtype=int)
                for child in adjacency_list[node] :
```

```
                    temp_array = train_classifiers(
                        ↪ classifiers , adjacency_list
                        ↪ , child , features ,
                        ↪ train_dataset_target ,
                        ↪ node_int_inverse_map ,
                        ↪ leaf_to_topic ,
                        ↪ classifier_map)
                    local_target[temp_array] = child
                    boolean_array = np.logical_or(
                        ↪ boolean_array , temp_array)
            local_features = features[boolean_array
                ↪ ,:]
            local_target = local_target[boolean_array
                ↪ ]
            classifiers[classifier_map[node]].fit(
                ↪ local_features , local_target)
            return boolean_array


def build_classifier_map(adjacency_list) :
        classifier_map = {}
        counter = 0
        till = len(adjacency_list)
        for i in range(till):
                if adjacency_list[i] != [] :
                        classifier_map[i] = counter
                        counter += 1
        return classifier_map

def change_index_value(x) :
        return leaf_to_topic[node_int_inverse_map[
            ↪ current_class]]

def predict_class(documents, classifiers , classifier_map ,
    ↪ leaf_to_topic , node_int_inverse_map ,
    ↪ count_vectorizer , tfidf_transformer):
        till = documents.shape[0]
        final_answer = np.zeros(shape=till)
        all_classifiers = sorted(list(classifier_map.keys
            ↪ ()))
        print(all_classifiers)

        for i in all_classifiers:
                temp_bool = (final_answer == i)
                final_answer[temp_bool] = np.array(
                    ↪ classifiers[classifier_map[i]].
```

```
                ↪ predict(documents[temp_bool,:]))
    for i in range(till):
            final_answer[i] = leaf_to_topic[
                ↪ node_int_inverse_map[final_answer[i
                ↪ ]]]
    return list(final_answer)
```