DS8001: Design of Algorithms and Programming for Massive Data

Research Project: Parallelization of End-to-End Ensemble Learning for Text Classification

Anika Tabassum
ID: 500865054
December 17, 2018

Background / Introduction

When we solve a machine learning problem by combining the results of several models or learning algorithms, it is known as ensemble learning. Ensemble learning almost always generates better predictions compared to a single model. As the volume of data continues to grow in recent years, ensemble learning has become more and more relevant and applicable. Ensemble models can be trained in parallel because they are trained independently.

A classical approach for ensemble learning methods is to use homogeneous models (ie models of the same type, such as decision trees), which created homogeneous ensembles. However, "heterogeneous ensembles" are also possible, when different types of models are used. The base learners have to be "as accurate as possible and as diverse as possible" for ensemble methods to be more accurate than its member individual learners [1].

Text classification is the process of extracting generic tags/categories from natural text, where the tags or categories are usually predefined. With the emergence of tons of text generating sources (social media, blogs, reviews) text classification has attained its place in a myriad of applications like product tagging, sentiment analysis, toxic comment detection, text source identification etc.

In this project we aim to train an ensemble learner for text classification. Over the years, a lot of machine learning techniques have been applied to text classification; combining the power of multiple classifiers gives us the ability to generate a better classifier. However, with the volume of text data growing, ensembling also leaves us with the challenge of scaling, and this is where the scope of parallelization comes in, which we explore in this project.

Goal

Our goal is to develop an end to end heterogeneous scalable ensemble learning algorithm and demonstrate the performance gain of using parallelism in certain portions of the code. We will use this ensemble learning algorithm for the training and evaluation of text classification. To be specific, we aim to train 3 shallow machine learning models (namely Logistic Regression, Naive Bayes and Random Forest) and 3 deep neural models (LSTMs with 3 different dropout rates) and execute the training and evaluation of these models in parallel. Aside from the actual training and evaluation, we will also need to do quite a bit of preprocessing of the data, which we also plan to parallelize. We will compare the performance of the parallelized portions of code with the sequential execution of the equivalent portions of code. Because of lack of access to a distributed cluster (Hadoop/Spark) and GPU server, we limit our implementation to a single 4-core CPU machine using Python's multiprocessing framework with the joblib wrapper.

For the purpose of text classification, we have chosen a problem where the task is to identify authors given text excerpts from books written by them. For this dataset, there are 3 predefined authors: Edgar Allan Poe, Mary Shelley, and HP Lovecraft. We have obtained the dataset from this Kaggle competition: https://www.kaggle.com/c/spooky-author-identification . The training dataset contains more than 18,000 records which, while not gigantic, is big enough to allow us to simulate a big data problem and experiment with parallelization.

Related literature

There has been quite a bit of work in the realm of parallelizing machine learning and ensemble algorithms over the last couple of decades.

PLANET [2], “a scalable distributed framework for learning tree models over large datasets”, defines “tree learning as a series of distributed computations, and implements each one using the MapReduce model of distributed computation”.

This thread [3] discusses a few ways of parallelizing machine learning algorithms such as using multiple CPU cores on a single CPU, using a GPU and using a cluster computing (Spark ML, Hadoop etc) infrastructure.

In [4], Prateek Khushalani and Dr. Victor Robin discuss some contexts and necessity of parallelizing machine learning algorithms. They point out that due to the number of algorithms available, their hyperparameters and the cross-validation, a data scientist might have to create thousands of models before reaching a good outcome, and parallelization could be handy in this sort of scenario.

In [5], the authors develop a parallel programming framework, one that is easily applied to many different learning algorithms, using Google's map-reduce paradigm. The Stanford article [6] also discusses the necessity of parallelizing machine learning algorithms as the data grows in size and complexity. This forum [7] discusses parallelism of machine learning algorithms in the context of CUDA and GPUs.

The popular machine learning framework scikit-learn provides off-the-shelf support to train some homogeneous ensemble algorithms (RandomForest and ExtraTreesClassifier) as well as cross validation in parallel. Another popular boosting library, XGBoost, has off-the-shelf support for distributing and parallelizing gradient boosted ensemble learning.

This project is not aiming to build any generic parallelization framework. It does however take inspiration from the above works and tries to build a scalable parallelization approach for heterogeneous ensemble learning which can be generalized to any classification or regression techniques.

Method

We use python’s multi-processing framework along with joblib wrapper, numpy, pandas, scikit learn and Keras to implement our end to end algorithm. **Figure 1** shows the high level flow diagram of our approach. The data contains only one feature column “text” which represents the text excerpt from any of the three authors. Note that we also set aside a small segment of the training data into a validation set, however for the purpose of this project (parallelization) that is not important. Below we describe briefly each of the important steps and the parallelization gains:

Preprocessing**:** After the training data is loaded (pandas dataframe), it first goes through a series of pre-processing steps like stemming, lemmatization and removal of stop words. This step has been parallelized by dividing the train data into 4 chunks and applying preprocessing on the chunks individually with 4 worker processes and then merging them back. For the sequential processing, it took 6.9 seconds while for the parallel processing it took 2.3 seconds.

Computing TFIDF**:** The preprocessed text data is then vectorized using scikit learn’s TFIDFVectorizer. After fitting the TFIDF vectorizer, we tried to parallelize vectorizing the sentences of the train data using fitted vectorizer by again dividing the data into 4 chunks and using 4 workers. However, parallelization in this case seemed to make the performance worse (4.66 seconds as opposed to 0.39 seconds in sequential). A possible reason could be that sklearn stores the TFIDF vectors in sparse matrix format, and in the parallel approach, during chunking and merging there is a lot of overhead going on to densify those and keeping track of the indices of the chunks.

Training Shallow Models: For part of the heterogeneous ensemble learning purpose, we build and train 3 shallow classifiers – Logistic Regression, Naive Bayes and Random Forest. We train them is parallel using 3 worker processes and the parallelism did bring us some performance gain, 3.4 seconds as opposed to 4.45 seconds in the sequential run. Note that, as opposed to the preprocessing and TFIDF cases, we don’t divide the data into chunks here. All three worker processes run with the full data.

The above steps conclude the shallow training portion of the ensemble learning. Let's go through the deep learning portion of the process.

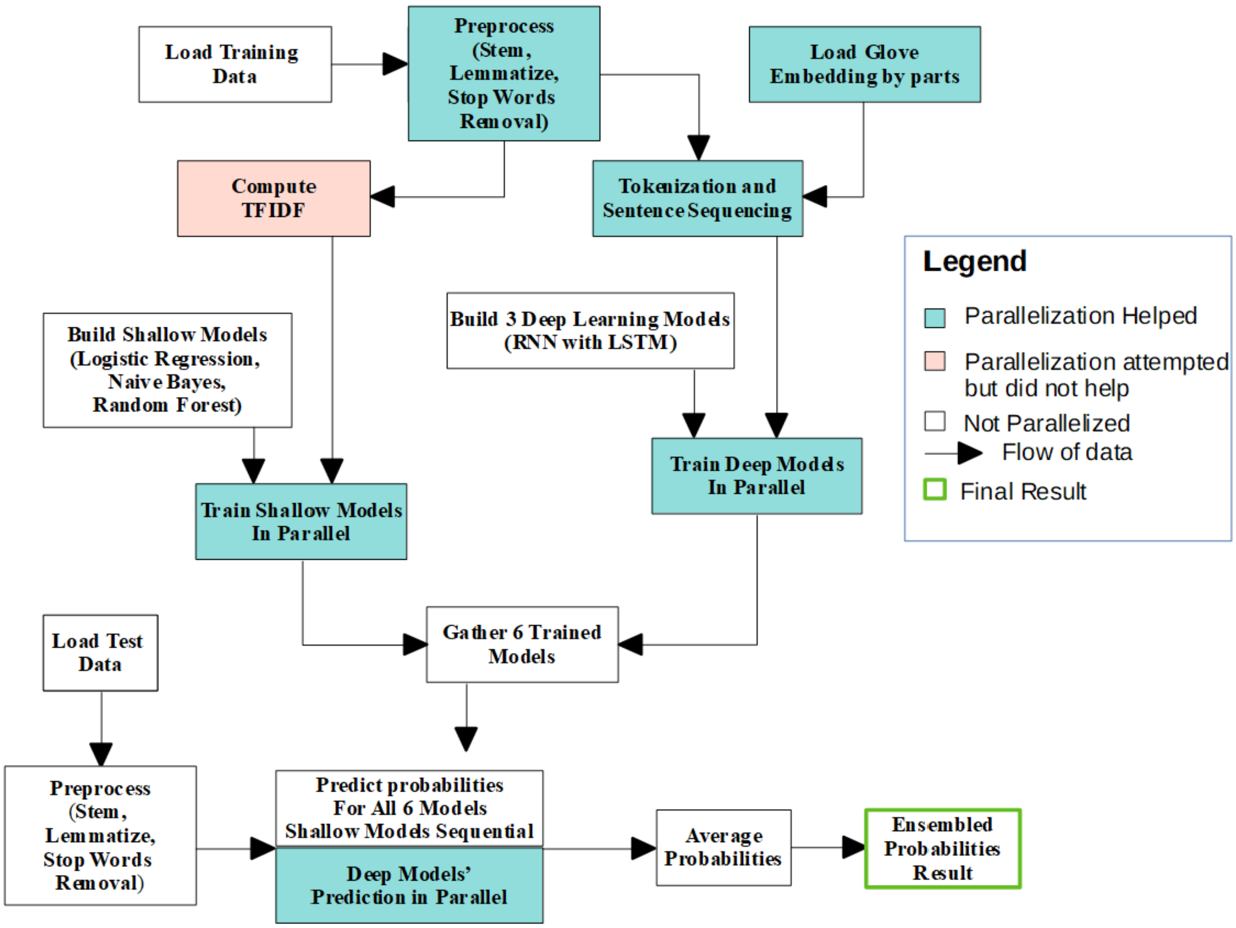

Figure 1: High level flow diagram of the ensemble training and testing

Loading the Glove Embedding: We use the well known Glove [8] embedding vectors for the purpose of vectorizing the words (and hence sentences) in an embedding space in the first layers of the deep neural models. We use the glove.840B.300d file which is over 5.5 GB in size. We divided the file into 4 equal chunks and loaded it with 4 parallel worker processes. This gained us a reasonable amount of performance gain – 111.46 seconds as opposed to 153.56 seconds in the sequential load. Note that unlike previous parallelization steps, this steps involve a lot of I/O (disk access).

Tokenization and Sequencing: We make use of Keras's tokenization and sequencing API methods to tokenize and vectorize our sentences before feeding it to the deep neural models. Again, we do this in parallel using 4 chunks of the data and 4 worker processes, and do have some performance gain – 24.19 seconds as opposed to 33.29 seconds in the sequential run.

Training the Deep Models: We use 3 deep neural networks each of which consists of an embedding layer, then an LSTM layer, followed by 3 fully connected layers and a softmax activation (3 classes) in the end. However, we use 3 different dropout rates (0.3, 0.4 and 0.5) to control the amount of regularization for our purpose of ensembling. We train these 3 models in parallel using 3 worker processes and we see that the performance gain is substantial – 809.98 seconds as opposed to 2127.93 seconds if the models are run sequentially – almost 260%. This is the step in the entire project that really stands out.

Ensembling the predictions: Once we have all 6 of our models fitted (3 shallow and 3 deep), we load our unlabeled test data, do all the necessary preprocessing on it, and predict the probabilities of the 3 classes using all 6 of them. Now we take the average probability of each class per record to calculate our final ensemble results, and this is where a final level of parallelism is applied. During prediction, we let the shallow models' prediction to happen sequentially since they are pretty fast, but we parallelize the predictions of the deep models, since they are not that fast. Thankfully, we do have some performance gain here as well – 14.85 seconds as opposed to 35.68 seconds in the full sequential run. This is another major performance gaining step in our process – almost 250%.

Results

Training Data Size: 15663 rows (after excluding the small validation percetange)

Step	**Sequential Execution Time**	**Parallel Execution Time**
Preprocessing	6.9 seconds	2.3 seconds (4 cores)
Shallow models training	4.45 seconds	3.4 seconds (3 cores)
Loading Glove Embedding	153.56 seconds	111.46 seconds (4 cores)
Tokenization and Sequencing	33.29 seconds	24.19 seconds (4 cores)
Deep models training	2127.93 seconds	809.98 seconds (3 cores)
Prediction result ensembling	35.68 seconds	14.85 seconds (3 cores)

Let's calculate the **Karp-Flatt** metric value for all the steps that we have parallelized.

For the preprocessing step $\psi = \frac{6.9}{2.3} = 3 \qquad e = \frac{\frac{1}{3} - \frac{1}{4}}{1 - \frac{1}{4}} = 0.11$

For the parallel training of the shallow models $\psi = \frac{4.45}{3.4} = 1.31 \qquad e = \frac{\frac{1}{1.31} - \frac{1}{3}}{1 - \frac{1}{3}} = 0.645$

For the parallel loading of the embeddings file $\psi = \frac{153.56}{111.46} = 1.378 \qquad e = \frac{\frac{1}{1.377} - \frac{1}{4}}{1 - \frac{1}{4}} = 0.634$

For the parallel tokenization and sequencing $\psi = \frac{33.29}{24.19} = 1.376 \qquad e = \frac{\frac{1}{1.376} - \frac{1}{4}}{1 - \frac{1}{4}} = 0.635$

For training the deep models in parallel $\psi = \frac{2127.93}{809.98} = 2.627 \qquad e = \frac{\frac{1}{2.627} - \frac{1}{3}}{1 - \frac{1}{3}} = 0.07$

For the parallel prediction with the deep models for ensembling $\psi = \frac{35.68}{14.45} = 2.4 \qquad e = \frac{\frac{1}{2.4} - \frac{1}{3}}{1 - \frac{1}{3}} = 0.125$

From the **Karp-Flatt** metric values, it seems that the deep model training step is the most effective parallelization step of all, followed by the preprocessing and the prediction ensembling, and then the rest. We perform some further experimentation of the deep model training step and the preprocessing step with varying data sizes to get an idea of the executions times for sequential vs parallel runs. **Figure 2** shows the growth trend of the sequential and parallel execution times – the deep model training on the left and the preprocessing on the right. It looks like that for lower data sizes, the effect of parallelism is not that evident (especially for the preprocessing case) and might actually be worse that the sequential execution, because of the overhead involved in managing the workers. However, as the data size grows, the performance gain with parallelism becomes quite evident. In the deep model training case, the sequential execution times seems to grow gradually with a slightly increasing slope, while the parallel execution grows linearly with a very small slope. In the preprocessing case, the sequential execution grows linearly with a reasonably big slope while the parallel execution time almost doesn't grow.

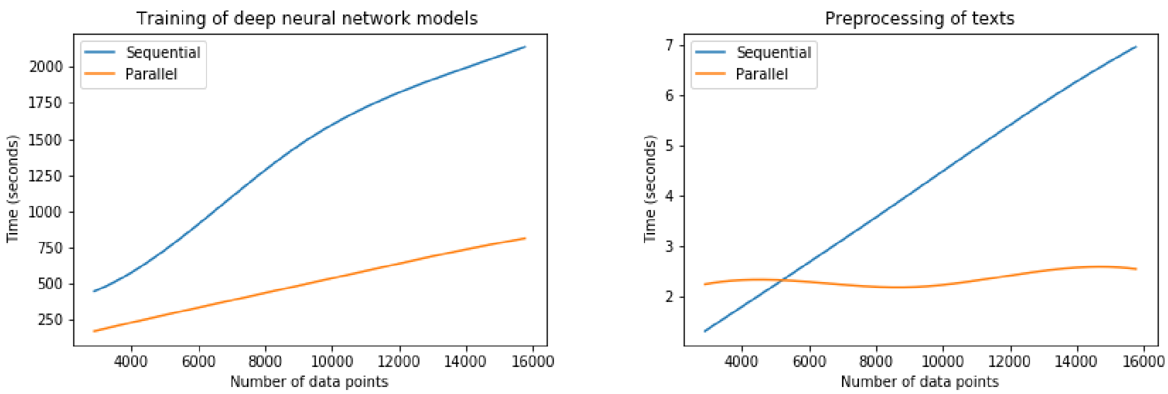

Figure 2: Comparison of sequential vs parallel execution

Summary

In this project, we have tried to parallelize segments of an end to end ensemble learning and evaluation process, and demonstrated that the parallelism indeed gains us some performance gain, and in some cases a lot of gain. We must emphasize that the focus of this project was not on improving the accuracy or tuning the machine learning models, rather on the parallelization aspects. We have showed that given some data and some models, we can craft the end to end ensemble learning framework in a way so that bits of it are processed in parallel to get better efficiency. There is definitely scope to explore further applicability of parallelism in this sort of ensemble learning. For example, the training and test data loading parts could be parallelized as well, and as we have new models, plenty of experimentation can be done as to see how exactly things should be parallelized in order to get maximum performance gain. One key takeaway from this project is that the effect of parallelism is truly visible only when the data is big enough. In the modern era of big data, this is certainly something to leverage. Notice that the data we are using is not too big in size, yet we are already seeing the gain in parallelizing. From our execution time measurements and from the sequential vs parallel execution figures in the results section, we can conclude that in a real scenario with gigantic amount of data, the gain with the kind of parallelism we have done could be significant and lead to a scalable approach for ensemble learning. Also, note that although we have worked on a text classification problem, the approach can be generalized to any mahcine learning problem where ensembling is used. As a final note, we have only used the power of a single multi-core machine for our parallelization purpose. We can definitely hope to improve performance much further by making use of a processors on multiple machines or using a cluster computing framework.

References

[1] Vadim Smolyakov, “Ensemble Learning to Imrpove Machine Learning Results”, 2017. [Online] Available: https://blog.statsbot.co/ensemble-learning-d1dcd548e936

[2] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. “PLANET: Massively parallel learning of tree ensembles with MapReduce”. In Proceedings of the 35th International Conference on Very Large Data Base (VLDB 2009), Lyon, France, 2009.

[3] “What is the best way to parallelize Machine Learning techniques?”, 2014. [Online] Available: https://www.researchgate.net/post/What_is_the_best_way_to_parallelize_Machine_Learning_techniques

[4] Prateek Khushalani, Victor Robin, “Parallel Processing of Machine Learning Algorithms”, 2018. [Online] Available: https://medium.com/dunnhumby-data-science-engineering/parallel-processing-of-machine-learning-algorithms-e1cff1151bef

[5] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, Kunle Olukotun. “Map-Reduce for Machine Learning on Multicore**”.** NIPS 2006

[6] Juan Batiz-Benet, Quinn Slack, Matt Sparks, Ali I Yahya . “Parallelizing Machine Learning Algorithms”. [Online] Available: http://cs229.stanford.edu/proj2010/BatizBenetSlackSparksYahya-ParallelizingMachineLearningAlgorithms.pdf

[7] Matthew Mayo, “Parallelism in Machine Learning: GPUs, CUDA, and Practical Applications”, 2016. [Online] Available: https://www.kdnuggets.com/2016/11/parallelism-machine-learning-gpu-cuda-threading.html

[8] Jeffrey Pennington, Richard Socher, Christopher D. Manning. “Glove: Global Vectors for Word Representation”, 2014. https://nlp.stanford.edu/projects/glove/

Appendix

A. Code

Below is the complete code of the project. The parallelization steps have been highlighted in yellow. The accompanying file DS8001_Code.pdf (created from a Jupyter Notebook file) has the code step by step with embedded documentation. The file also contains the prerequisites and instructions for a successful run of the program.

```
import pandas as pd
import numpy as np
import math
import nltk

from nltk.stem import WordNetLemmatizer
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from nltk import word_tokenize

from sklearn.naive_bayes import MultinomialNB, GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.preprocessing import LabelEncoder
from scipy.sparse import hstack as sp_hstack, vstack as sp_vstack
from scipy.interpolate import make_interp_spline, BSpline

from joblib import Parallel, delayed

import time

from keras.models import Sequential
from keras.layers.recurrent import LSTM, GRU
from keras.layers.core import Dense, Activation, Dropout, Layer, K
from keras.layers.embeddings import Embedding
from keras.layers.normalization import BatchNormalization
from keras.utils import np_utils
from keras.layers import GlobalMaxPooling1D, Conv1D, MaxPooling1D, Flatten,
Bidirectional, SpatialDropout1D
from keras.preprocessing import sequence, text
from keras.callbacks import EarlyStopping, Callback

import pickle
import matplotlib.pyplot as plt
%matplotlib inline
```

```
test_set = pd.read_csv('test.csv')
training_set = pd.read_csv('train.csv')

df_train, df_val = train_test_split(training_set, stratify=training_set['author'],
random_state=20, test_size=0.2, shuffle=True)

x_train = df_train['text']
y_train = df_train['author']
x_val = df_val['text']
y_val = df_val['author']

label_enc = LabelEncoder()
y_train = label_enc.fit_transform(y_train)
y_val = label_enc.transform(y_val)

def next_chunk(data):
    for i in range(4):
        yield data[math.ceil(i * len(data) / 4):math.ceil((i + 1) * len(data) / 4)]

def multiclass_logloss(actual, predicted, eps=1e-15):
    """Multi class version of Logarithmic Loss metric.
    :param actual: Array containing the actual target classes
    :param predicted: Matrix with class predictions, one probability per class
    """
    # Convert 'actual' to a binary array if it's not already:
    if len(actual.shape) == 1:
        actual2 = np.zeros((actual.shape[0], predicted.shape[1]))
        for i, val in enumerate(actual):
            actual2[i, val] = 1
        actual = actual2

    clip = np.clip(predicted, eps, 1 - eps)
    rows = actual.shape[0]
    vsota = np.sum(actual * np.log(clip))
    return -1.0 / rows * vsota

nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')

ls = LancasterStemmer()
lem = WordNetLemmatizer()
stop_words = stopwords.words('english')

def normalize(text):
   words = word_tokenize(text)
```

```
   words = [w for w in words if not w in stop_words]

   stemw = [ls.stem(w) for w in words]

   # 2- Lemmatization
   lemw = [lem.lemmatize(w) for w in stemw]
   return ' '.join(lemw)

def get_preprocessed_data(data):
    data = data.apply(normalize)
    return data

start_preprocess = time.time()
x_train_preprocessed = get_preprocessed_data(x_train)
end_preprocess = time.time()
time_preprocess_seq = end_preprocess - start_preprocess
print('Time taken by sequential pre_processing of training data: {}
seconds'.format(time_preprocess_seq))

start_preprocess = time.time()
x_train_preprocessed = np.hstack(Parallel(n_jobs=4, backend='loky', verbose=10)\
                            (delayed(get_preprocessed_data)(data) for data in
next_chunk(x_train)))

end_preprocess = time.time()
time_preprocess_par = end_preprocess - start_preprocess
print('Time taken by parallelizing pre_processing of training data: {}
seconds'.format(time_preprocess_par))

x_val_preprocessed = get_preprocessed_data(x_val)

tfidf_vectorizer = TfidfVectorizer(min_df=3,  max_features=None,
            strip_accents='unicode', analyzer='word',token_pattern=r'\w{1,}',
            ngram_range=(1, 3), use_idf=1,smooth_idf=1,sublinear_tf=1,
            stop_words = 'english')
tfidf_vectorizer.fit(x_train_preprocessed)

start_tfidf = time.time()
x_train_prep_tfidf = tfidf_vectorizer.transform(x_train_preprocessed)
end_tfidf = time.time()
time_tfidf_seq = end_tfidf - start_tfidf
print('Time taken by sequential tfidf calculation of training data: {}
seconds'.format(time_tfidf_seq))

start_tfidf = time.time()
x_train_prep_tfidf = sp_vstack(Parallel(n_jobs=4, backend='loky', verbose=10)\
```

```
                             (delayed(tfidf_vectorizer.transform)(data) for data in
next_chunk(x_train_preprocessed)))

end_tfidf = time.time()
time_tfidf_par = end_tfidf - start_tfidf
print('Time taken by parallel tfidf calculation of training data: {}
seconds'.format(time_tfidf_par))

x_val_prep_tfidf = tfidf_vectorizer.transform(x_val_preprocessed)

logreg_model = LogisticRegression(C=11.0)
logreg_model.fit(x_train_prep_tfidf, y_train)

nb_model = MultinomialNB()
nb_model.fit(x_train_prep_tfidf, y_train)

rfmodel = RandomForestClassifier(n_estimators=100, random_state=1,
min_samples_leaf=3, n_jobs=1)

rfmodel.fit(x_train_prep_tfidf, y_train)

models = [logreg_model, rfmodel, nb_model]

def fit_and_evaluate_model(model, x_train, y_train, x_val, y_val):
    model.fit(x_train, y_train)
    probs = model.predict_proba(x_val)
    score = model.score(x_val, y_val)
    return probs, score

start_ensemble = time.time()
results = []
for model in models:
    results.append(fit_and_evaluate_model(model, x_train_prep_tfidf, y_train,
x_val_prep_tfidf, y_val))
end_ensemble = time.time()
time_ensemble_seq = end_ensemble - start_ensemble
print('Time taken by sequential run of ensemble model: {}
seconds'.format(time_ensemble_seq))

start_ensemble = time.time()
results = Parallel(n_jobs=3, backend='loky', verbose=0)\
                             (delayed(fit_and_evaluate_model)(model,
x_train_prep_tfidf, y_train, x_val_prep_tfidf, y_val) \
                              for model in models)
end_ensemble = time.time()
time_ensemble_par = end_ensemble - start_ensemble
```

```
print('Time taken by parallel run of ensemble model: {}
seconds'.format(time_ensemble_par))

# This is a utility function and the code is from
# https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/6.1-
using-word-embeddings.ipynb
def read_embeddings(file):
    embeddings_index = {}
    with open(file, encoding="utf8") as f:
       for line in f:
           values = line.split(' ')
           word = values[0]
           coefs = np.asarray(values[1:], dtype='float32')
           embeddings_index[word] = coefs
    return embeddings_index

start_embedding = time.time()
read_embeddings('glove.840B.300d.txt')
end_embedding = time.time()
time_embedding_seq = end_embedding - start_embedding
print('Time taken by sequential embeddings read: {}
seconds'.format(time_embedding_seq))

start_embedding = time.time()
files = ['glove_part1.txt', 'glove_part2.txt', 'glove_part3.txt',
'glove_part4.txt']
indices = Parallel(n_jobs=4, backend='loky', verbose=10)\
                            (delayed(read_embeddings)(file) for file in files)
embeddings_index = {}
for index in indices:
    embeddings_index.update(index)
end_embedding = time.time()

#time_embedding_par = end_embedding - start_embedding
print('Time taken by parallel embeddings read: {}
seconds'.format(time_embedding_par))
print('Found %s word vectors.' % len(embeddings_index))

max_len = 100
#max_words = 10000
# using keras tokenizer here
tokenizer = text.Tokenizer(num_words=None)
tokenizer.fit_on_texts(x_train_preprocessed)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

```
# The code for embedding_matrix is taken from:
# https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/6.1-using-word-embeddings.ipynb
embedding_matrix = np.zeros((len(word_index) + 1, 300))
for word, i in word_index.items():
   embedding_vector = embeddings_index.get(word)
   if embedding_vector is not None:
       embedding_matrix[i] = embedding_vector

def get_sequence_data(data):
   data = tokenizer.texts_to_sequences(data)
   # zero pad the sequences
   data = sequence.pad_sequences(data, maxlen=max_len)
   return data

start_sequence = time.time()
x_train_sequence = get_sequence_data(x_train_preprocessed)
end_sequence = time.time()
time_sequence_seq = end_sequence - start_sequence
print('Time taken by sequential tokenization and sequencing: {} seconds'.format(end - start))

start_sequence = time.time()
x_train_sequence = np.vstack(Parallel(n_jobs=4, backend='loky', verbose=10)\
                            (delayed(get_sequence_data)(data) for data in next_chunk(x_train_preprocessed)))
end_sequence = time.time()
time_sequence_par = end_sequence - start_sequence
print('Time taken by parallel tokenization and sequencing: {} seconds'.format(end - start))

x_val_sequence = get_sequence_data(x_val_preprocessed)

y_train_enc = np_utils.to_categorical(y_train)
y_val_enc = np_utils.to_categorical(y_val)

def nn_model(dropout=0.3):
    model = Sequential()
    model.add(Embedding(len(word_index) + 1,
                        300,
                        weights=[embedding_matrix],
                        input_length=max_len,
                        trainable=False))
    model.add(SpatialDropout1D(0.3))
    model.add(LSTM(100, dropout=dropout, recurrent_dropout=dropout))
```

```
    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(dropout))

    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(dropout))

    model.add(Dense(3))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
    return model

def fit_and_evaluate_nn_model(model, x_train, y_train, x_val, y_val,
batch_size=512):
    model.fit(x_train, y=y_train, batch_size=batch_size, epochs=10, verbose=1)
    probs = model.predict(x_val)
    _, score = model.evaluate(x_val, y_val)
    return probs, score

def get_nn_models():
    return [nn_model(0.3), nn_model(0.4), nn_model(0.5)]

start_ensemble_nn = time.time()
nn_results = []
models_seq = get_nn_models()
for model in models_seq:
    nn_results.append(fit_and_evaluate_nn_model(model, x_train_sequence,
y_train_enc, x_val_sequence, y_val_enc))
end_ensemble_nn = time.time()
time_ensemble_nn_seq = end_ensemble_nn - start_ensemble_nn
print('Time taken by sequential training of 3 models: {}
seconds'.format(time_ensemble_nn_seq))

start_ensemble_nn = time.time()
models_par = get_nn_models()
nn_results = Parallel(n_jobs=3, backend='loky', verbose=1)\
                            (delayed(fit_and_evaluate_nn_model)(model,
x_train_sequence, y_train_enc,
                                                                x_val_sequence,
y_val_enc) \
                             for model in models_par)
end_ensemble_nn = time.time()
time_ensemble_nn_par = end_ensemble_nn - start_ensemble_nn
print('Time taken by parallelizing training of 3 models: {}
seconds'.format(time_ensemble_nn_par))
```

```
x_test = test_set['text']
x_test_preprocessed = get_preprocessed_data(x_test)
x_test_prep_tfidf = tfidf_vectorizer.transform(x_test_preprocessed)
x_test_sequence = get_sequence_data(x_test_preprocessed)

start_ensemble_result = time.time()
ensemble_probs = models[0].predict_proba(x_test_prep_tfidf)
for model in models[1:]:
    ensemble_probs += model.predict_proba(x_test_prep_tfidf)
for model in models_par:
    ensemble_probs += model.predict(x_test_sequence)
ensemble_probs /= 6
end_ensemble_result = time.time()
time_ensemble_result_seq = end_ensemble_result - start_ensemble_result
print('Time taken by sequential ensemble result calculation: {}
seconds'.format(time_ensemble_result_seq))

start_ensemble_result = time.time()

ensemble_probs = models[0].predict_proba(x_test_prep_tfidf)
for model in models[1:]:
    ensemble_probs += model.predict_proba(x_test_prep_tfidf)

ensemble_results_deep = Parallel(n_jobs=3, backend='loky', verbose=1)\
                            (delayed(model.predict)(x_test_sequence) \
                             for model in models_par)
for result in ensemble_results_deep:
    ensemble_probs +=result
ensemble_probs /= 6

end_ensemble_result = time.time()
time_ensemble_result_par = end_ensemble_result - start_ensemble_result
print('Time taken by parallel ensemble result calculation: {}
seconds'.format(time_ensemble_result_par))

sizes = [3000, 6000, 9000, 12000]

times_nn_seq = []
for i, size in enumerate(sizes):
    start_ensemble_nn = time.time()
    models_seqq = get_nn_models()
    for model in models_seqq:
        fit_and_evaluate_nn_model(model, x_train_sequence[:size],
y_train_enc[:size], x_val_sequence, y_val_enc)
    end_ensemble_nn = time.time()
    times_nn_seq.append(end_ensemble_nn - start_ensemble_nn)
```

```
    print('Time taken by sequential training of 3 models with size {}: {}
seconds'.format(size, times_nn_seq[i]))

times_nn_parr = []
for i, size in enumerate(sizes):
    start_ensemble_nn = time.time()
    models_parr = get_nn_models()
    Parallel(n_jobs=3, backend='loky', verbose=1)\
                            (delayed(fit_and_evaluate_nn_model)(model,
x_train_sequence[:size], y_train_enc[:size],
                                                                 x_val_sequence,
y_val_enc) \
                             for model in models_parr)
    end_ensemble_nn = time.time()
    times_nn_parr.append(end_ensemble_nn - start_ensemble_nn)
    print('Time taken by parallel training of 3 models with size {}: {}
seconds'.format(size, times_nn_parr[i]))

times_nn_seq.append(time_ensemble_nn_seq)
times_nn_parr.append(time_ensemble_nn_par)
sizes.append(len(x_train))

def interplotate_smooth(x, y):
    x_new = np.linspace(min(x) - 100, max(x) + 100, 300)

    spl = make_interp_spline(x, y, k=3)
    y_new = spl(x_new)
    return x_new, y_new

plt.plot(*interplotate_smooth(sizes, times_nn_seq), label='Sequential')
plt.plot(*interplotate_smooth(sizes, times_nn_parr), label='Parallel')
plt.xlabel('Number of data points')
plt.ylabel('Time (seconds)')
plt.title('Training of deep neural network models ')
plt.legend()
plt.savefig('neural_seq_vs_par.png')
plt.show()

times_prep_seq = []
for i, size in enumerate(sizes[:4]):
    start_preprocess = time.time()
    temp = get_preprocessed_data(x_train[:size])
    end_preprocess = time.time()
    times_prep_seq.append(end_preprocess - start_preprocess)
    print('Time taken by sequential preprocessing with size {}: {}
seconds'.format(size, times_prep_seq[i]))
```

```
times_prep_seq.append(time_preprocess_seq)

times_prep_parr = []
for i, size in enumerate(sizes[:5]):
    start_preprocess = time.time()
    temp = np.hstack(Parallel(n_jobs=4, backend='loky', verbose=10)\
                            (delayed(get_preprocessed_data)(data) for data in 
next_chunk(x_train)))
    end_preprocess = time.time()
    times_prep_parr.append(end_preprocess - start_preprocess)
    print('Time taken by parallel preprocessing with size {}: {} 
seconds'.format(size, times_prep_parr[i]))

plt.plot(*interplotate_smooth(sizes, times_prep_seq), label='Sequential')
plt.plot(*interplotate_smooth(sizes, times_prep_parr), label='Parallel')
plt.xlabel('Number of data points')
plt.ylabel('Time (seconds)')
plt.title('Preprocessing of texts')
plt.legend()
plt.savefig('prep_seq_vs_par.png')
plt.show()
```