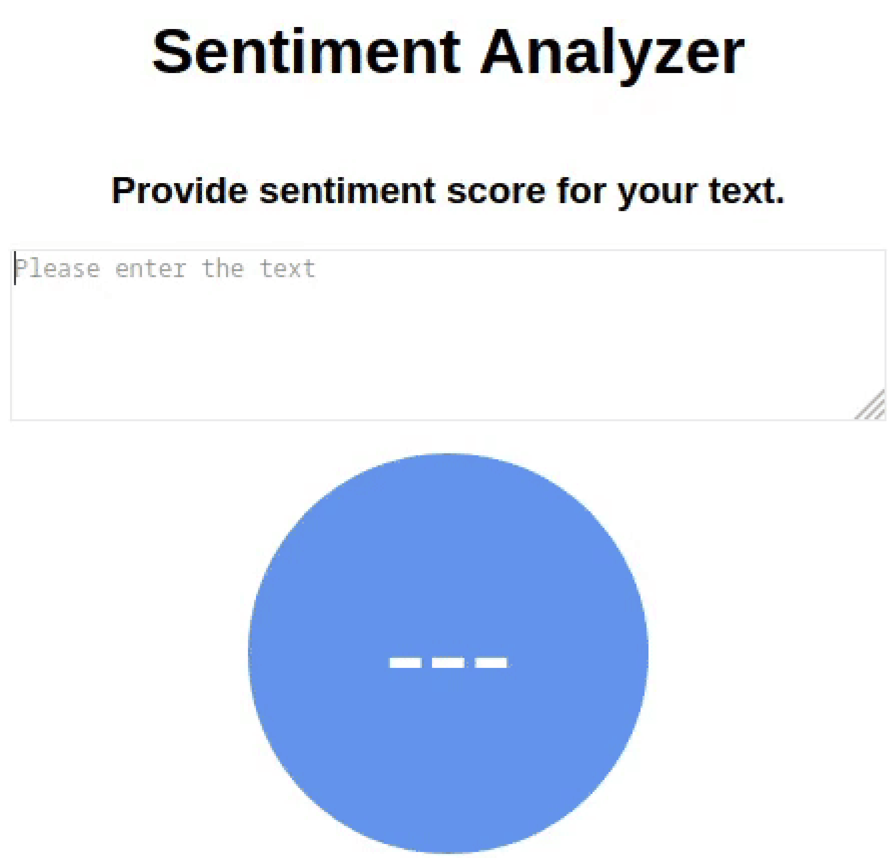
Sentiment Analyzer with BERT (build, tune, deploy)

Brief description of how I developed sentiment analyzer. It covers text preprocessing, model building, tuning, API, frontend creation and containerization.

Zuzanna Deutschman Jul 24, 2020 · 4 min read ★



Dataset

I used the dataset published by The Stanford NLP Group. I merged two files, namely 'dictionary.txt' including 239,232 text fragments and 'sentiment_labels.txt' containing the sentiment scores assigned to the various text fragments.

Text preprocessing with regular expressions

To clean the text, I usually use a bunch of functions containing regular expressions. In `common.py` you can find all of them, for example `remove_nonwords` described below:

```python
def remove_nonwords(df, column):
    """
    Replace non-words from the beginning and end of the string
    Parameters
    ----------
    df : data frame
    column : column name with input text

    Returns
    -------
    df : data frame with cleaned text
    """
    reg = r"^[^a-zA-Z]*|[^a-zA-Z]*$"
    df[column] = df[column].str.replace(reg, "")
    return df
```

remove_nonwords.py hosted with ♡ by GitHub view raw

Similar functions were used for empty rows, special signs, numbers and html code removal.

After text cleaning, it's time for BERT embeddings creation. For that purpose, I used bert-as-service. It is very simple and consists of only 3 steps: download a pre-trained model, start the BERT service and use client for sentence encodings of specified length.

```
def BERT_embeddings(train_x, test_x, valid_x, valid_y):
    bc = BertClient(ip='ip')
    train_seq_x = bc.encode(list(train_x.values))
    test_seq_x = bc.encode(list(test_x.values))
    valid_seq_x = bc.encode(list(valid_x.values))

    train_x = pd.DataFrame(data=train_seq_x, index=train_x.index)
    test_x = pd.DataFrame(data=test_seq_x, index=test_x.index)
    valid_x = pd.DataFrame(data=valid_seq_x, index=valid_x.index)
    return train_x, test_x, valid_x
```

BERT_embeddings.py hosted with ♡ by GitHub view raw

There are multiple parameters that can be setup, when running a service. For example, to define `max_seq_len` , I calculated 0.9 quantile of train data length.

```
def max_sequence_length(train_x):
    return int(train_x.str.split().str.len().quantile(0.9))
```

max_seq_len.py hosted with ♡ by GitHub view raw

Preprocessed data has a form of data frame containing 768 features. For full code, please go to `nlp_preprocess.py` .

Model building with Keras

In this part, we build and train the model on different parameters. Let's assume we want 5-layers neural network as below. We will parametrize batch_size, number of epochs, number of nodes in the first 4 dense layers and 5 dropout layers.

```
def build_model(train_x, batch_size=128,
                dense_1_nodes=256, dense_2_nodes=256, dense_3_nodes=256, dense_4_nodes=256,
                dropout_1_size=0.1, dropout_2_size=0.1, dropout_3_size=0.1, dropout_4_size=0.1,
```

```
    model = Sequential()

    # The Input Layer :
    model.add(Dense(dense_1_nodes, kernel_initializer='normal',input_dim = train_x.shape[1], act
    model.add(Dropout(rate=dropout_1_size))

    # The Hidden Layers :
    model.add(Dense(dense_2_nodes, kernel_initializer='normal',activation='relu'))
    model.add(Dropout(dropout_2_size))

    model.add(Dense(dense_3_nodes, kernel_initializer='normal',activation='relu'))
    model.add(Dropout(dropout_3_size))

    model.add(Dense(dense_4_nodes, kernel_initializer='normal',activation='relu'))
    model.add(Dropout(dropout_4_size))

    # The Output Layer :
    model.add(Dense(units = 1, activation="sigmoid"))
    model.add(Dropout(dropout_5_size))
    model.compile(loss='mean_squared_error', optimizer='rmsprop', metrics = ['mse', 'mae'])
    print(model.summary)
    return model
```

build_model.py hosted with ♡ by GitHub view raw

```
def model_training(train_x, train_y, valid_x, valid_y, epochs, batch_size,
                   dense_1_nodes, dense_2_nodes, dense_3_nodes, dense_4_nodes,
                dropout_1_size, dropout_2_size, dropout_3_size, dropout_4_size, dropout_5_size):
    model = build_model(train_x, batch_size,
                         dense_1_nodes, dense_2_nodes, dense_3_nodes, dense_4_nodes,
                dropout_1_size, dropout_2_size, dropout_3_size, dropout_4_size, dropout_5_size)
    history = model.fit(train_x.values, train_y.values, epochs=epochs, batch_size=batch_size, ver
    model.save('model.h5')
    return model, history
```

model_training.py hosted with ♡ by GitHub view raw

Model tuning with Sacred

Now we can tune the parameters. We will use `sacred` module. Key points here are:

1. Create an Experiment and add Observer

First we need to create an experiment and observer that logs all kinds of information. It's very simple!

```
from sacred import Experiment
from sacred.observers import MongoObserver

ex = Experiment()
ex.observers.append(MongoObserver(
    url='mongodb://192.168.1.2:27017'))
```

create_experiment.py hosted with ♡ by GitHub view raw

2. Define the main function

The `@ex.automain` decorator defines and runs the main function of the experiment when we run the Python script.

```
@ex.automain
def main(epochs, batch_size,
         dense_1_nodes, dense_2_nodes, dense_3_nodes, dense_4_nodes,
         dropout_1_size, dropout_2_size, dropout_3_size, dropout_4_size, dropout_5_size):

    train_x, train_y, test_x, test_y, valid_x, valid_y = load_model()
    model, history = model_training(train_x, train_y, valid_x, valid_y, epochs=epochs, batch_siz
                                    dense_1_nodes=dense_1_nodes, dense_2_nodes=dense_2_nodes, de
                                    dropout_1_size=dropout_1_size, dropout_2_size=dropout_2_size
    model_testing(test_x, test_y, model, history)
```

main.py hosted with ♡ by GitHub view raw

3. Add the Configuration parameters

We will define them through Config Scope.

```
@ex.config
def cfg():
```

```
    epochs=30
    batch_size=128
    dense_1_nodes=256
    dense_2_nodes=256
    dense_3_nodes=256
    dense_4_nodes=256
    dropout_1_size=0.1
    dropout_2_size=0.1
    dropout_3_size=0.1
    dropout_4_size=0.1
    dropout_5_size=0.3
```

cfg.py hosted with ♡ by GitHub view raw

4. Add metrics

In our case here I want to know the **MAE** and **MSE**. We can use the Metrics API for that.

```
def model_testing(test_x, test_y, model, history):
    pred_y = model.predict(test_x)
    np.savetxt("data/pred_y.csv", pred_y, delimiter=",")
    mae = mean_absolute_error(pred_y, test_y)
    mse = mean_squared_error(pred_y, test_y)
    ex.log_scalar("MAE", mae)
    ex.log_scalar("MSE", mse)
```

model_testing.py hosted with ♡ by GitHub view raw

5. Run the experiment

Functions from the previous steps are stored in `model_experiment.py` script. In order to run our exepriment for bunch of parameters, we create and run `run_sacred.py`. For all possible permutations, MAE and MSE will be saved in MongoDB.

```
from model_experiment import ex
import itertools

batch_size_values =[64, 128]
dense_1_nodes =[64, 128]
dense_2_nodes = [64, 128]
dense_3_nodes= [64, 128]
dense_4_nodes = [64, 128]
```

```
dropout_1_size = [0.1. 0.3]
dropout_2_size= [0.1, 0.3]
dropout_3_size = [0.1, 0.3]
dropout_4_size = [0.1, 0.3]
dropout_5_size = [0.1, 0.3]
epochs_values = [30]


for epochs, batch_size, dense_1_nodes, dense_2_nodes, dense_3_nodes, dense_4_nodes, dropout_1_si
    in itertools.product(epochs_values, batch_size_values, dense_1_nodes, dense_2_nodes, dense_3
    ex.run(config_updates={'dropout_5_size': dropout_5_size, 'batch_size': batch_size, 'epochs':
                           'dense_1_nodes': dense_1_nodes, 'dense_2_nodes': dense_2_nodes,
                           'dense_3_nodes': dense_3_nodes, 'dense_4_nodes': dense_4_nodes,
                           'dropout_1_size': dropout_1_size, 'dropout_2_size': dropout_2_size,
                           'dropout_3_size': dropout_3_size, 'dropout_4_size': dropout_4_size
                           })
```

run_sacred.py hosted with ♡ by GitHub view raw

The best result I got is 9% of MAE score. That means that our sentiment analyzer works pretty good. We can check it with `model_inference` function.

```
def model_inference(model, sentence):
    bc = BertClient()
    sentence = bc.encode(sentence)
    score = model.predict(sentence)
    pred_y = np.loadtxt(os.path.normpath("data/pred_y.csv"), delimiter=",")
    min_score, max_score = min_max_sentiment(pred_y)
    norm_score = (score - min_score)/(max_score - min_score)
    return norm_score
```

model_inference.py hosted with ♡ by GitHub view raw

Please note that the score is normalized so that outlier values can be also obtained. After model is saved, we can build a Web API!

Web API creation with Flask

Now we want to create an API that runs the code in the function and displays the returned result in the browser.

```
@app.route('/score', methods = ['PUT'])
@cross_origin()
def score():
    print(request)
    text = request.json['text']
    df = pd.DataFrame(columns=['text']).append({'text': text}, ignore_index=True)
    df = text_cleaning(df, 'text')
    preproc_text = df.iloc[0][0]
    score = float(model_inference(model, [preproc_text]))
    return {"score": score}
```

app.py hosted with ♡ by GitHub view raw

The syntax `@app.route('/score', methods=['PUT'])` lets Flask know that the function, `score`, should be mapped to the *endpoint/score*. The `methods` list is a keyword argument that tells us what kind of HTTP requests are allowed. We'll be using `PUT` requests to receive sentences from a user. In function `score`, we get a score in dictionary form, since it can be easily converted to a JSON string. Full code is available in `api.py`.

Frontend

For web interface, three files were created:

- `index.html` provides the basic structure of the site: title, description, input text area and circle with score.
- `style.css` is used to style the website.
- `index.js` provides interactivity. It is responsible for reading user input, handling API requests and presenting calculated score. Three main functions here are:

```
function textChangedHandler(text) {
    console.log(text)
    fetch("http://127.0.0.1:5000/score", {
```

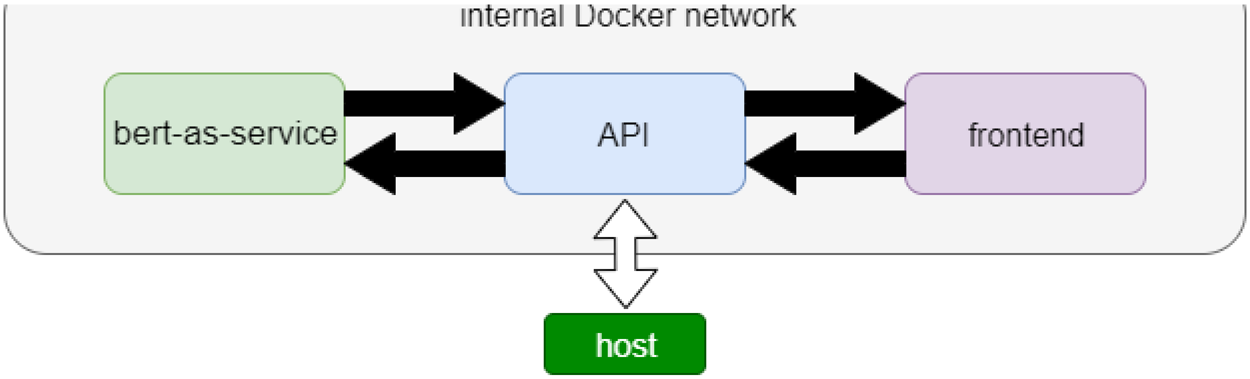
```
        method: "PUT",
        body: JSON.stringify({"text": text}),
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'},
    }).then(parseResponse)
}

const textChanged = debounce(textChangedHandler, 250);

function parseResponse(response){
    console.log(response)
    response.json().then(handleJSON)
}

function handleJSON(response){
    let score = (response.score * 100).toFixed(0)+'%'
    console.log(score)
    document.getElementById("score").textContent = score
    document.getElementById("circle").style.backgroundColor = hsv2hex({h: response.score*120, s:
}
```

index.js hosted with ♡ by GitHub view raw

For gradient **HSV** model was used. Saturation and Value are constants. Hue corresponds to score value. Changing hue in range [0;120] yields smooth colour change from red to yellow to green.

Docker containerization

The brilliance of Docker is that, once you package an application and all its dependencies into container, you ensure it will run in any environment. It is generally recommended to separate areas of concern by using one service per container. In my small app there are 3 parts that should be combined: bert-as-service, application and frontend. The tool that helps you build Docker images and run containers is **Docker Compose**.



Steps that we need to do to dockerize our code:

- Create separate folders for bert-as-service, api and frontend,
- Put there relevant files,
- Add `requirenments.txt` and `Dockerfile` to each folder. The first file should cover all needed libraries that will be installed via command in the second file. Its format is described in docker documentation
- Create `docker-compose.yaml` in the 3 folders directory. Define the 3 services that make up the app in this file, so they can be run together in an isolated environment.

Now we are ready to build and run our application! Please see the sample outputs below.

As usual, please feel free to view the full code on my Gitlab.

Projects · Zuzanna / Sentiment Analysis with BERT

GitLab.com

gitlab.co

Data Science Deep Learning NLP Deployment Machine Learning