TECH-GB.3332 Final Project (Fall 2020)

Bryan Clark & Vamsi Ponnapalli

12/13/2020

Contents

Overview

In this term project, you will deploy Deep Learning models to build a classification model using RapidMiner to predict the sentiment of consumers towards US airlines based on their reviews expressed in the form of tweets. If you strongly prefer to use some other DL-based software/frameworks instead of RapidMiner, such as TensorFlow or PyTorch, let me know before starting the work. This is a group project, and you should work on it in the groups that you have formed already.

Python Libraries

```python
# fetch data
import requests
from io import StringIO

# core
import numpy as np
import pandas as pd

# preprocessing
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

```
# baseline algorithms
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

# deep learning
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout
from tensorflow.keras.losses import mean_absolute_error as tf_mae
from tensorflow.keras.optimizers import Adam
import tensorflow as tf

# evaluation
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_validate
from sklearn.metrics import mean_absolute_error as skl_mae

# print out
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

1. Fetch Data

The data is provided to you in two versions:

1. The original version of the tweets (and their sentiments) is located at https://drive.google.com/file/d/1atyRH5Yz7TU-2ziyZknfd7ib2LLwYeuv/view?usp=sharing.
2. The preprocessed version of the tweets is located at https://drive.google.com/file/d/1c96crlNZr7XiF3-9lmZ1nEJaY3MHTTz5/view?usp=sharing, where text preprocessing and pre-training of the text embeddings of the tweets using autoencoders have already been done to make your life simpler. This preprocessed version contains the sentiments about the tweets in column 1 of the spreadsheet (either positive (1) or negative (0)) and the 8-dimenisonal pre-trained embeddings of the tweets (in columns 2 – 9 of the spreadsheet).

I recommend that you use the preprocessed version of the tweets since it will save you a lot of preprocessing work to build these embeddings that is non-trivial. However, if you like challenges, you can do preprocessing and building the embeddings using autoencoders yourself and, therefore, work directly with the "raw" tweets. As a "reward" for this extra work, you will be awarded 10 extra points (the max score of this project is 100) if you preprocess tweets yourself.

```
# url of dataset
google_drive_file_url = 'https://drivesds.google.com/file/d/' \
    + '1c96crlNZr7XiF3-9lmZ1nEJaY3MHTTz5/view?usp=sharing'

def fetch_google_drive_csv(google_drive_file_url):

    file_id = google_drive_file_url.split('/')[-2]
    download_url = 'https://drive.google.com/uc?export=download&id=' + file_id
    url = requests.get(download_url)
    csv_raw = StringIO(url.text)
    return pd.read_csv(csv_raw)

data = fetch_google_drive_csv(google_drive_file_url)

data.head()
```

```
##    sentiment  dimension1  dimension2  ...  dimension6  dimension7  dimension8
## 0          1   -0.400418    0.293417  ...   -0.034476    0.042176   -0.429317
## 1          1   -0.454608   -0.194998  ...    0.064868    0.072154    0.629457
## 2          0   -0.515892   -0.120781  ...   -0.155029   -0.306803    0.694974
## 3          1    0.047770   -0.230509  ...   -0.229259   -0.835945    0.294148
## 4          0   -0.574353   -0.132517  ...   -0.338480    0.202040   -0.100443
##
## [5 rows x 9 columns]
```

```
data.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 55524 entries, 0 to 55523
## Data columns (total 9 columns):
##  #   Column      Non-Null Count  Dtype
## ---  ------      --------------  -----
##  0   sentiment   55524 non-null  int64
##  1   dimension1  55524 non-null  float64
##  2   dimension2  55524 non-null  float64
##  3   dimension3  55524 non-null  float64
##  4   dimension4  55524 non-null  float64
##  5   dimension5  55524 non-null  float64
##  6   dimension6  55524 non-null  float64
##  7   dimension7  55524 non-null  float64
##  8   dimension8  55524 non-null  float64
## dtypes: float64(8), int64(1)
## memory usage: 3.8 MB
```

2. Preprocess Data

Your task is to predict the score of the sentiment (positive or negative) between 0 and 1 based on the embeddings of the tweets specified in columns 2 – 9 of the pre-possessed spreadsheet (or the original tweets if you decided to work with the raw tweeting data). To evaluate the performance of your model, please split the dataset into the train set and the test set in the 0.8:0.2 ratio and use cross-validation to calculate the prediction performance.

2.1 Train/Test Split

```
target = 'sentiment'

data = data.astype('float32')

X_train, X_test, y_train, y_test = train_test_split(
    data.drop(target, axis=1), # predictors
    data[target], # target
    test_size=0.2,
    random_state=90
)
```

```
X_train.shape
```

```
## (44419, 8)
```

```
X_test.shape
```

```
## (11105, 8)
```

2.2 Cross-Validation

2.2.1 Cross-Validation Splits

```
cv_splits = KFold(n_splits=5, random_state=90, shuffle=True)

fold_num = 1
for train, test in cv_splits.split(X_train, y_train):
    print(f'Fold #{fold_num}: Train shape: {X_train.iloc[train].shape},',
          f'Test shape: {X_train.iloc[test].shape}')
    fold_num += 1
```

```
## Fold #1: Train shape: (35535, 8), Test shape: (8884, 8)
## Fold #2: Train shape: (35535, 8), Test shape: (8884, 8)
## Fold #3: Train shape: (35535, 8), Test shape: (8884, 8)
## Fold #4: Train shape: (35535, 8), Test shape: (8884, 8)
## Fold #5: Train shape: (35536, 8), Test shape: (8883, 8)
```

2.2.2 Keras Cross-Validation Grid Search

```
def nn_grid_search(model):

    param_grid = {
        'epochs': [50],
        'batch_size':[25, 50, 100, 250, 500]
    }

    grid = GridSearchCV(
        estimator=model,
        param_grid=param_grid,
        scoring='neg_mean_absolute_error',
        n_jobs=-1,
        cv=cv_splits
    )

    grid_result = grid.fit(X_train, y_train)

    # summarize results
    print(f"Best: {-grid_result.best_score_:.4f}; {grid_result.best_params_:}")
    means = -grid_result.cv_results_['mean_test_score']
    stds = grid_result.cv_results_['std_test_score']
    params = grid_result.cv_results_['params']
    for mean, stdev, param in zip(means, stds, params):
        print(f"{mean:.4f} +/- {stdev:.4f} with: {param}")

    return grid_result
```

3. Neural Networks

You can use any neural network model you like for this classification task. In particular, you may start with a simple single fully connected network as a “baseline” and then try to use more complex models, including

CNN and RNN based models, to achieve better performance results than this simple baseline model. Your goal is to reach the mean absolute error of at least 0.48, which should not be too difficult. If you want to be more ambitious, you can try to reach the mean absolute error of 0.47 (medium difficulty), or even 0.46 (this is difficult). The higher accuracy you get, the more points you will be awarded.

In addition to the simple NN baseline mentioned above, you should also build another basic baseline, such as a logistic regression model (similar to the one we used in the RapidMiner Lab done in the class) and compare the performance results of your DL-based model with that baseline. The expectation is that the more sophisticated DL-model should outperform simple baselines.

3.1 Baseline Model Performance

3.1.1 Logistic Regression

```
model_lr = LogisticRegression()
cv_results_lr = cross_validate(
    model_lr,
    X_train,
    y_train,
    scoring='neg_mean_absolute_error',
    cv=cv_splits
)
cv_lr_mu = -cv_results_lr['test_score'].mean()
cv_lr_sd = cv_results_lr['test_score'].std()

print(f'Logistic Regression MAE: {cv_lr_mu:.4f} +/- {cv_lr_sd:.4f}\n')
```

```
## Logistic Regression MAE: 0.4366 +/- 0.0041
```

```
_ = model_lr.fit(X_train, y_train)
```

Time to execute this code chunk: 0.32 s

3.1.2 Random Forest

```
model_rf = RandomForestClassifier(n_estimators=500)
cv_results_rf = cross_validate(
    model_rf,
    X_train,
    y_train,
    scoring='neg_mean_absolute_error',
    cv=cv_splits
)
cv_rf_mu = -cv_results_rf['test_score'].mean()
cv_rf_sd = cv_results_rf['test_score'].std()

print(f'Random Forest MAE: {cv_rf_mu:.4f} +/- {cv_rf_sd:.4f}\n')
```

```
## Random Forest MAE: 0.4174 +/- 0.0057
```

```
_ = model_rf.fit(X_train, y_train)
```

Time to execute this code chunk: 4.55 s

3.1.3 Simple Neural Network

```
def create_model_nn0():

    # model configuration
    loss_function = tf_mae
    optimizer = Adam()

    model = Sequential([
        Dense(32, activation='relu', input_dim=8),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(loss=loss_function,
                  optimizer=optimizer)

    return model

model_nn_0 = KerasClassifier(build_fn=create_model_nn0, verbose=0)
cv_results_nn_0 = nn_grid_search(model_nn_0)
```

```
## Best: 0.4525; {'batch_size': 500, 'epochs': 50}
## 0.4623 +/- 0.0185 with: {'batch_size': 25, 'epochs': 50}
## 0.4604 +/- 0.0231 with: {'batch_size': 50, 'epochs': 50}
## 0.4610 +/- 0.0191 with: {'batch_size': 100, 'epochs': 50}
## 0.4619 +/- 0.0155 with: {'batch_size': 250, 'epochs': 50}
## 0.4525 +/- 0.0245 with: {'batch_size': 500, 'epochs': 50}
```

Time to execute this code chunk: 2.34 s

3.2 Deep Learning

We are going to look at constructing two deep learning nerual network architectures. For each ones, we are going to keep the `Adam` optimizer from the simple model, `mean absolute error` as the loss function, and the number of epochs at 100 with an early stopping monitor. Additionally, we will maintain the `relu` activation function for hidden layers and `sigmoid` activation for the output layer.

- The `Adam` optimizer is chosen as it is the recommended optimizer from the book as it is an iteration of optimizers developed before it (mainly `RMSProp`), but we could go back to `Stochastic Gradient Descent` to simplify the learning process.
- `Mean absolute error` is the loss function for the assignment to reduce.
- `ReLU` is chosen as it overcomes some of the limitations of the `sigmoid` activation function and is one of the most commonly used activation functions for hidden layers.
- `Sigmoid` activation function is used for the final output layer as it will restrict values to a range [0,1], which will correspond to a prediction for the probability of having positive sentiment.

We will test out different options for batch size, the number of hidden layers, and the number of neurons in each layer. Each model will use 50 `epochs` and we could further optimize this as needed if desired with minimal updates to the code.

3.2.1 DL Model #1

For our first deep learning model, we are going to focus on increasing the number of hidden layers with a modest amount of neurons per layer. We will have a fully connected netowrk with 8 hidden layers of 8 neurons each. No dropout or batch normalization will be applied. The thought-process is to dramatically increase the complexity of the network to see how the performance changes.

```
def create_model_nn1():

    # model configuration
    loss_function = tf_mae
    optimizer = Adam()

    model = Sequential([
        Dense(8, activation='relu', input_dim=8),
        Dense(8, activation='relu'),
        Dense(8, activation='relu'),
        Dense(8, activation='relu'),
        Dense(8, activation='relu'),
        Dense(8, activation='relu'),
        Dense(8, activation='relu'),
        Dense(8, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(loss=loss_function,
                  optimizer=optimizer)

    return model


model_nn_1 = KerasClassifier(build_fn=create_model_nn1, verbose=0)
cv_results_nn_1 = nn_grid_search(model_nn_1)
```

```
## Best: 0.4459; {'batch_size': 250, 'epochs': 50}
## 0.4607 +/- 0.0157 with: {'batch_size': 25, 'epochs': 50}
## 0.4600 +/- 0.0142 with: {'batch_size': 50, 'epochs': 50}
## 0.4514 +/- 0.0187 with: {'batch_size': 100, 'epochs': 50}
## 0.4459 +/- 0.0160 with: {'batch_size': 250, 'epochs': 50}
## 0.4602 +/- 0.0162 with: {'batch_size': 500, 'epochs': 50}
```

Time to execute this code chunk: 2.56 s

3.2.2 DL Model #2

For the second deep learning model, we will dial back on some of the layers and add more neurons in each layer (starting at 64 and reducing the number in each subsequent layer). We will also add batch normalization with each hidden layer to allow layers to reduce the impact each layer has on the next as well as add a form of regularization to the model. This is still a fairly simple network design, so if this performs well, we have options to increase the complexity to further optimize our loss function.

```
def create_model_nn2():

    # model configuration
    loss_function = tf_mae
    optimizer = Adam()

    model = Sequential([
        Dense(64, activation='relu', input_dim=8),
        BatchNormalization(),
        Dense(32, activation='relu'),
        BatchNormalization(),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(loss=loss_function,
                  optimizer=optimizer)

    return model

model_nn_2 = KerasClassifier(build_fn=create_model_nn2, verbose=0)
cv_results_nn_2 = nn_grid_search(model_nn_2)
```

```
## Best: 0.4212; {'batch_size': 500, 'epochs': 50}
## 0.4271 +/- 0.0046 with: {'batch_size': 25, 'epochs': 50}
## 0.4220 +/- 0.0055 with: {'batch_size': 50, 'epochs': 50}
## 0.4239 +/- 0.0029 with: {'batch_size': 100, 'epochs': 50}
## 0.4231 +/- 0.0041 with: {'batch_size': 250, 'epochs': 50}
## 0.4212 +/- 0.0054 with: {'batch_size': 500, 'epochs': 50}
```

Time to execute this code chunk: 4.1 s

4. Evaluation

After you build your neural network, apply the trained deep learning model to the test set and evaluate its performance using the accuracy measures.

4.1 Holdout Performance

```
# baseline algo predictions
preds_lr = model_lr.predict_proba(X_test)[:,1]
preds_rf = model_rf.predict_proba(X_test)[:,1]

# predict test test for NN models
preds_nn_0 = cv_results_nn_0.best_estimator_.predict_proba(X_test)[:,1]
preds_nn_1 = cv_results_nn_1.best_estimator_.predict_proba(X_test)[:,1]
preds_nn_2 = cv_results_nn_2.best_estimator_.predict_proba(X_test)[:,1]

# # # evaluation of test predictions
mae_test_lr = skl_mae(y_test, preds_lr)
mae_test_rf = skl_mae(y_test, preds_rf)
mae_test_nn_0 = skl_mae(y_test, preds_nn_0)
mae_test_nn_1 = skl_mae(y_test, preds_nn_1)
mae_test_nn_2 = skl_mae(y_test, preds_nn_2)

test_results_df = pd.DataFrame({
    'Model': ['logistic regression', 'random forest', 'simple NN', 'DL #1', 'DL #2'],
    'MAE': [mae_test_lr, mae_test_rf, mae_test_nn_0, mae_test_nn_1, mae_test_nn_2]
}).sort_values('MAE').reset_index(drop=True)

test_results_df
```

```
##                   Model       MAE
## 0                DL #2  0.420449
## 1                DL #1  0.428250
## 2            simple NN  0.440916
## 3        random forest  0.459881
## 4  logistic regression  0.489189
```

4.2 Conclusions

When testing each model on the holdout data, we see that the `DL #2` considerably outperforms the rest of the models (they rest may vary depending on the run). We could attempt to improve on `DL #2` by further tweaking the number of neurons, number of hidden layers, optimizer, and/or more explicit forms of regularization.