Sentiment Analysis with IMDB dataset

This notebook provides a simple straight-forward way to achieve 90% accuracy on IMDB dataset.

Load Data

In [1]:

```
import utils

# Function for loading imdb dataset
def load_imdb():
    train, test = utils.get_imdb_dataset()
    TEXT_COL, LABEL_COL = 'text', 'sentiment'
    return (
        train[TEXT_COL], train[LABEL_COL],
        test[TEXT_COL], test[LABEL_COL])
```

In [2]:

```
train_text, train_label, test_text, test_label = load_imdb()
```

```
data already available, skip downloading.
imdb loaded successfully.
```

In [3]:

```
# Check Shape, should not throw exceptions
for data in train_text, train_label, test_text, test_label:
    assert data.shape == (25000,)
```

Prepare Data

Build Vectorizer

In [4]:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(
    min_df=2, # ignore word that only appears in 1 document
    ngram_range=(1, 2), # consider both uni-gram and bi-gram
)
```

In [5]:

```
# Learn (fit) and transform text into vector
train_x = tfidf_vectorizer.fit_transform(train_text)

# Convert label to 0 and 1 (optional)
train_y = train_label.apply(lambda x: 1 if x == 'pos' else 0)
```

In [6]:

```
# Check the shape
print('Training x shape:', train_x.shape)
print('Training y shape:', train_y.shape)
```

```
Training x shape: (25000, 438350)
Training y shape: (25000,)
```

In [7]:

```
# Expect 12500 for 1 and 0, instead of pos and neg
train_y.value_counts()
```

Out[7]:

```
1    12500
0    12500
Name: sentiment, dtype: int64
```

In [8]:

```
# Apply the same transformer to validation set as well
test_x = tfidf_vectorizer.transform(test_text)
test_y = test_label.apply(lambda x: 1 if x == 'pos' else 0)
```

In [9]:

```
# Sanity check
assert test_x.shape == train_x.shape
assert test_y.shape == train_y.shape
```

Dimensionality Reduction

In this notebook, `SelectKBest` from `sklearn` is used to reduce dimensionality and using `f_classif` to help up pick up k best features (word).

In [10]:

```
from sklearn.feature_selection import SelectKBest
```

In [11]:

```
DIM = 20000 # Dimensions to keep, a hyper parameter

# Create a feature selector
# By default, f_classif algorithm is used
# Other available options include mutual_info_classif, chi2, f_regression
 etc.

selector = SelectKBest(k=20000)
```

In [12]:

```
# The feature selector also requires information from labels
# Fit on training data
selector.fit(train_x, train_y)
```

Out[12]:

```
SelectKBest(k=20000, score_func=<function f_classif at 0x00000229D46E379
8>)
```

In [13]:

```
# Apply to both training data and testing data
train_x = selector.transform(train_x)
test_x = selector.transform(test_x)
```

In [14]:

```
# Sanity check
assert train_x.shape == (25000, 20000)
assert test_x.shape == (25000, 20000)
```

Build a MLP Model

Muti-Layer Perceptron model, aka Feed Forward Network, is the most basic neural network structure, but is used in quite a lot of place as it is very robust. It is true that deep networks are usually more powerful, but they are usually more data hungry. In this coding demostration, for local computation efficieny, I didn't use much data, hence a MLP model may works better.

In [1]:

```
from tensorflow.keras.models import Model
from tensorflow.python.keras.layers import Input, Dense, Dropout
```

In [16]:

```
def build_mlp_model(input_dim, layers, output_dim, dropout_rate=0.2):
    # Input layer
    X = Input(shape=(input_dim,))

    # Hidden layer(s)
    H = X
    for layer in layers:
        H = Dense(layer, activation='relu')(H)
        H = Dropout(rate=dropout_rate)(H)

    # Output layer
    activation_func = 'softmax' if output_dim > 1 else 'sigmoid'

    Y = Dense(output_dim, activation=activation_func)(H)
    return Model(inputs=X, outputs=Y)
```

In [17]:

```
hyper_params = {
    'learning_rate': 1e-3,  # default for Adam
    'epochs': 1000,
    'batch_size': 64,
    'layers': [64, 32],
    'dim': DIM,
    'dropout_rate': 0.5,
}
```

In [18]:

```
mlp_model = build_mlp_model(
    input_dim=hyper_params['dim'],
    layers=hyper_params['layers'],
    output_dim=1,
    dropout_rate=hyper_params['dropout_rate'],
)

mlp_model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 20000)]           0         
_________________________________________________________________
dense (Dense)                (None, 64)                1280064   
_________________________________________________________________
dropout (Dropout)            (None, 64)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080      
_________________________________________________________________
dropout_1 (Dropout)          (None, 32)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33        
=================================================================
Total params: 1,282,177
Trainable params: 1,282,177
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [19]:

```
from tensorflow.keras.optimizers import Adam
```

In [20]:

```
mlp_model.compile(
    optimizer=Adam(lr=hyper_params['learning_rate']),
    loss='binary_crossentropy',
    metrics=['acc'],
)
```

Callbacks

Two common callbacks were used here: `EarlyStopping` and `ModelCheckpoint`. The first is used to prevent overfitting and the second is used to keep track of the best models we got so far.

In [21]:

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint
```

In [22]:

```
early_stoppping_hook = EarlyStopping(
    monitor='val_loss',  # what metrics to track
    patience=2,  # maximum number of epochs allowed without imporvement on monitored metrics
)

CPK_PATH = 'model_cpk.hdf5'    # path to store checkpoint

model_cpk_hook = ModelCheckpoint(
```

```
    CPK_PATH,
    monitor='val_loss',
    save_best_only=True,  # Only keep the best model
)
```

Train the Model, Hope for the Best

In [23]:

```
his = mlp_model.fit(
    train_x,
    train_y,
    epochs=10,
    validation_data=[test_x, test_y],
    batch_size=hyper_params['batch_size'],
    callbacks=[early_stoppping_hook, model_cpk_hook],
)

print('Training finished')
```

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/10
25000/25000 [==============================] - 9s 356us/sample - loss: 0.
3634 - acc: 0.8491 - val_loss: 0.2378 - val_acc: 0.9017
Epoch 2/10
25000/25000 [==============================] - 9s 343us/sample - loss: 0.
1422 - acc: 0.9508 - val_loss: 0.2493 - val_acc: 0.8998
Epoch 3/10
25000/25000 [==============================] - 9s 344us/sample - loss: 0.
0838 - acc: 0.9740 - val_loss: 0.2944 - val_acc: 0.8957
Training finished
```

Evaluation

Load the best model and do evaluation:

In [24]:

```
# Load the model checkpoint
mlp_model.load_weights(CPK_PATH)

# Accuracy on validation
mlp_model.evaluate(test_x, test_y)
```

```
25000/25000 [==============================] - 4s 155us/sample - loss: 0.
2378 - acc: 0.9017
```

Out[24]:

```
[0.23776556309223176, 0.90168]
```