项目 3

PROJECT 3

基于 LSTM 的影评情感分析

本项目基于 LSTM，使用分类数据集（Large Movie Review Dataset，LMRD）训练情感分析模型，实现移动端的文本情感推断设计。

3.1 总体设计

本部分包括系统整体结构图和系统前后端流程图。

3.1.1 系统整体结构图

系统整体结构如图 3-1 所示。

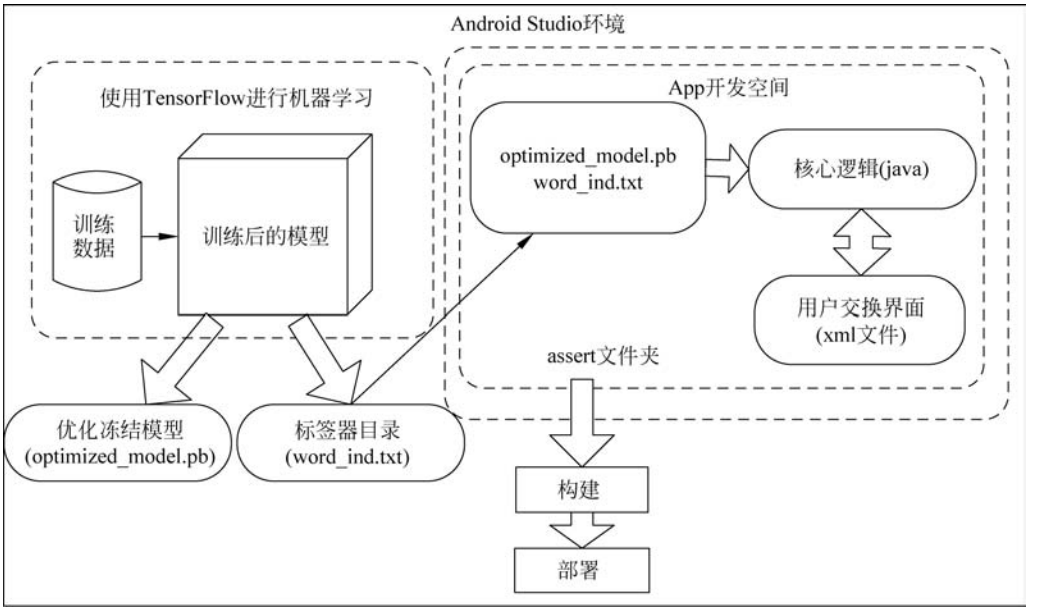


图 3-1　系统整体结构图

3.1.2　系统前后端流程图

系统前端流程如图 3-2 所示，系统后端流程如图 3-3 所示。

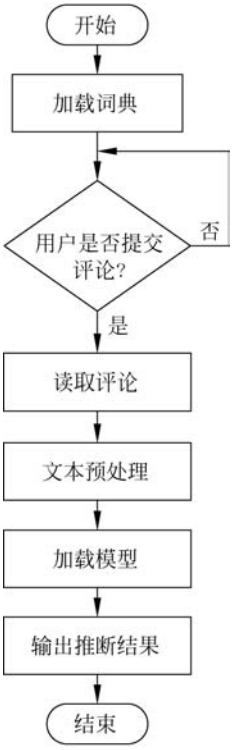


图 3-2　系统前端流程图

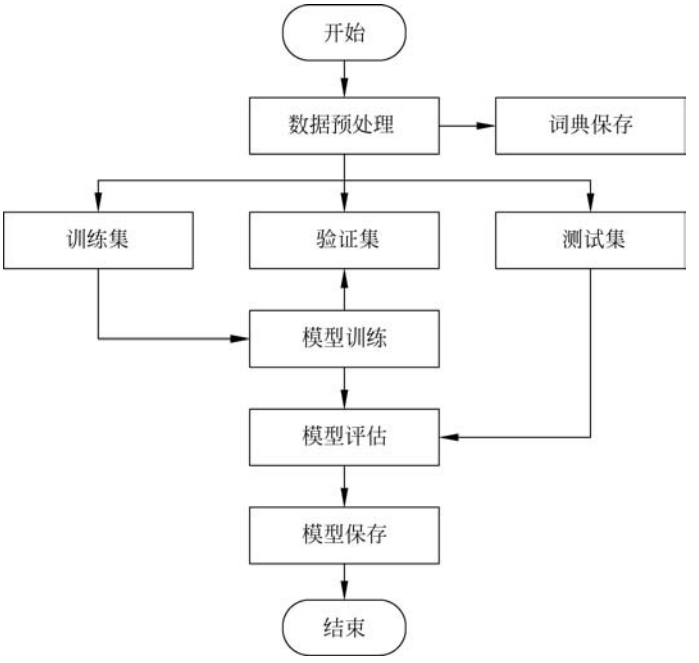


图 3-3　系统后端流程图

3.2　运行环境

本部分包括 Python 环境、TensorFlow 环境和 Android 环境。

3.2.1　Python 环境

需要 Python 3.6 及以上配置，用 Anaconda 创建虚拟环境 MRSA（全标为 Movie Review Sentiment Analysis），完成所需 Python 环境的配置。

打开 Anaconda Prompt，输入命令：

```
conda create -n MRSA python=3.6
```

创建 MRSA 虚拟环境。

3.2.2 TensorFlow 环境

打开 Anaconda Prompt,激活所创建的 MRSA 虚拟环境,输入命令:

```
activate MRSA
```

安装 CPU 版本的 TensorFlow,输入命令:

```
pip install - upgrade -- ignore - installed tensorflow
```

安装完毕。

其他相关依赖包,包括 Keras、Re、Pickle、Fire、Pandas、Numpy,其安装方式和 TensorFlow 类似,直接在虚拟环境中 pip install package_name 或者 conda install package_name 即可完成。

3.2.3 Android 环境

安装 Android Studio 新建 Android 项目,打开 Android Studio,依次选择 File→New→New Project→Empty Activity→Next。

Name 定义 Movie Review Analysis,Save location 为项目保存的地址,可自行定义,Minimum API 为该项目能够兼容 Android 手机的最低版本,选择 16。单击 Finish 按钮,新建项目完成。App/build.gradle 里的内容有任何改动后,Android Studio 都会弹出信息提示。单击 Sync Now 按钮或 图标,同步该配置,"成功"表示配置完成。

3.3 模块实现

本项目包括 5 个模块:数据预处理、模型构建及训练、模型保存、词典保存和模型测试。下面分别给出各模块的功能介绍及相关代码。

3.3.1 数据预处理

本部分包括数据集合并、数据清洗、文本数值化和数据集划分。

1. 数据集合并

数据集下载地址为 http://ai.stanford.edu/~amaas/data/sentiment/。斯坦福大学提供的情感分类数据集中了 25 000 条电影评论用于训练,25 000 条用于测试。先将这 50 000 条数据合并,并保存为.csv 文件格式,相关代码如下:

```
#导入原始数据
train_review_files_pos = os.listdir(path + 'train/pos/')
review_dest.append(path + 'train/pos/')
```

```
train_review_files_neg = os.listdir(path + 'train/neg/')
review_dest.append(path + 'train/neg/')
test_review_files_pos = os.listdir(path + 'test/pos/')
review_dest.append(path + 'test/pos/')
test_review_files_neg = os.listdir(path + 'test/neg/')
review_dest.append(path + 'test/neg/')
#将标签合并
sentiment_label = [1] * len(train_review_files_pos) + \
                  [0] * len(train_review_files_neg) + \
                  [1] * len(test_review_files_pos) + \
                  [0] * len(test_review_files_neg)
#将所有评论合并
review_train_test = ['train'] * len(train_review_files_pos) + \
                    ['train'] * len(train_review_files_neg) + \
                    ['test'] * len(test_review_files_pos) + \
                    ['test'] * len(test_review_files_neg)
#将合并后的数据保存为.csv 格式
df = pd.DataFrame()
df['Train_test_ind'] = review_train_test
df['review'] = reviews
df['sentiment_label'] = sentiment_label
df.to_csv(path + 'processed_file.csv', index = False)
```

合并后的.csv 文件如图 3-4 所示。

```
data = pd.read_csv(path + 'processed_file.csv')
print('数据大小为：',data.shape)
data.head()
```

数据大小为： (50000, 3)

	Train_test_ind	review	sentiment_label
0	train	bromwell high is a cartoon comedy it ran at th...	1
1	train	homelessness or houselessness as george carlin...	1
2	train	brilliant over acting by lesley ann warren bes...	1
3	train	this is easily the most underrated film inn th...	1
4	train	this is not the typical mel brooks film it was...	1

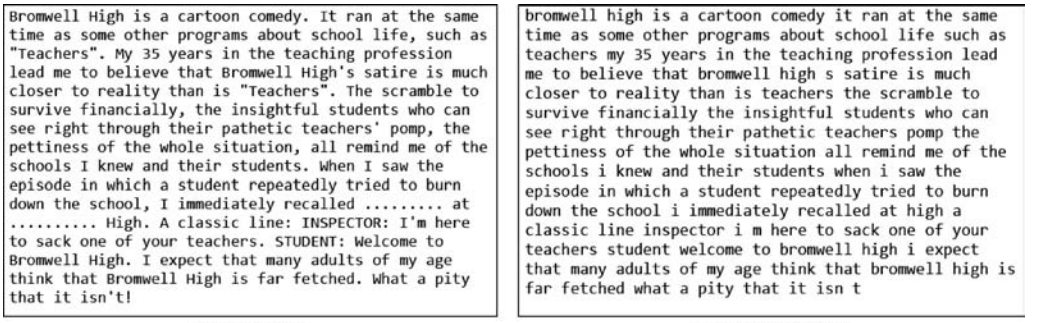
图 3-4　合并后的.csv 文件

2. 数据清洗

文本中一些非相干因素会影响最后模型的精度，采用正则表达式将所有标点符号去除，将大写字母转换成小写字母，相关代码如下：

```
def text_clean(text):
    #将所有大写字母转换成小写字母,并去除标点符号
    letters = re.sub("[^a-zA-z0-9\s]", " ",text)
    words = letters.lower().split()
    text = " ".join(words)
    return text
```

数据清洗结果如图 3-5 所示。



(a) 原始文本　　(b) 处理后的文本

图 3-5　数据清洗结果

3. 文本数值化

文本中每个单词对应唯一的索引(token),依据索引将文本数值化。Keras tokenizer 通过采集前 50 000 个常用词,转换为数字索引或标记。为了处理方便,对于文本长度大于 1000 的评论,只取前 1000 个单词;若评论长度不足 1000,则在评论开始使用 0 填充。相关代码如下:

```
#采集前 50000 个常用词,把单词转换为数字索引或标记
max_features = 50000
tokenizer = Tokenizer(num_words = max_features, split = ' ')
tokenizer.fit_on_texts(df['review'].values)
X = tokenizer.texts_to_sequences(df['review'].values)
X_ = []
for x in X:
    x = x[:1000]
    X_.append(x)
X_ = pad_sequences(X_)
```

数值化结果如图 3-6 所示。

4. 数据集划分

将数据集划分为训练集、验证集及测试集,比例分别为 70%、15%和 15%。相关代码如下:

```
y = df['sentiment_label'].values
```

```
records_processed 50000
[[    0     0     0 ...     4    11    16]
 [    0     0     0 ...  1173 22081    75]
 [    0     0     0 ...     9     1  1912]
 ...
 [    0     0     0 ...   167    32   363]
 [    0     0     0 ...   681     1  9109]
 [    0     0     0 ...    34   318    11]]
```

图 3-6　数值化结果

```
index = list(range(X_.shape[0]))
np.random.shuffle(index)
train_record_count = int(len(index) * 0.7)
validation_record_count = int(len(index) * 0.15)
train_indices = index[:train_record_count]
validation_indices = index[train_record_count:train_record_count +
                     validation_record_count]
test_indices = index[train_record_count + validation_record_count:]
X_train, y_train = X_[train_indices], y[train_indices]
X_val, y_val = X_[validation_indices], y[validation_indices]
X_test, y_test = X_[test_indices], y[test_indices]
```

划分后的数据集如图 3-7 所示。

```
x_train = np.load(path + 'X_train.npy')
x_val = np.load(path + 'X_val.npy')
x_test = np.load(path + 'X_test.npy')
print('训练数据集大小：',x_train.shape)
print('验证数据集大小：',x_val.shape)
print('测试数据集大小：',x_test.shape)
```

```
训练数据集大小：  (35000, 1000)
验证数据集大小：  (7500, 1000)
测试数据集大小：  (7500, 1000)
```

图 3-7　划分后的数据集

3.3.2　模型构建及训练

将数据加载进模型之后，需要定义模型结构、优化损失函数和性能指标。这里定义了两种结构进行训练，一是基于 BasicLSTM 的网络；二是基于 MultiRNN 的网络。

1. 定义模型结构

首先，构建一个简单的 LSTM 版本递归神经网络（BasicLSTM），并在输入层后面放一个嵌入层。嵌入层的单词向量使用预先训练好的 100 维 Glove 向量初始化，该图层被定义为 trainable（可训练的），这样，该单词向量嵌入层就可以根据训练数据自行更新。隐藏状态的维度和单元状态的维度也是 100。

其次，为获得文本中更多正确信息，进一步定义多层递归神经网络（MultiRNN），共有三层，每层单元状态的维度分别是 100、200、100。定义嵌入层的相关代码如下：

```
#定义嵌入层
with tf.variable_scope('embedding'):
    self.emb_W = tf.get_variable('word_embeddings', [self.n_words, self.embedding_dim],
      initializer = tf.random_uniform_initializer( - 1, 1, 0), trainable = True,
                              dtype = tf.float32)
    self.assign_ops = tf.assign(self.emb_W, self.emd_placeholder)
    self.embedding_input = tf.nn.embedding_lookup(self.emb_W, self.X, "embedding_input")
    print(self.embedding_input)
    self.embedding_input = tf.unstack(self.embedding_input, self.sentence_length, 1)
#定义网络结构
with tf.variable_scope('LSTM_cell'):
    #定义 BasicLSTM
    self.cell = tf.nn.rnn_cell.BasicLSTMCell(self.hidden_states)
    #定义 MultiRNN
    #num_units = [100, 200, 100]
    #self.cells = [tf.nn.rnn_cell.BasicLSTMCell(num_unit) for num_unit in num_units]
    #self.cell = tf.nn.rnn_cell.MultiRNNCell(self.cells)
```

2. 优化损失函数

使用二进制交叉熵损失训练模型,并在损失函数中加入正则化以防止出现过拟合,同时使用 Adam(Adaptivemoment estimation)优化器训练模型,用精确度作为性能指标。相关代码如下:

```
self.l2_loss = tf.nn.l2_loss(self.w, name = "l2_loss")
self.scores = tf.nn.xw_plus_b(self.output[ - 1],self.w,self.b, name = "logits")
self.prediction_probability = tf.nn.sigmoid(self.scores, name = 'positive_sentiment_
probability')     #计算属于 1 类的概率
self.predictions = tf.round(self.prediction_probability, name = 'final_prediction')
self.losses = tf.nn.sigmoid_cross_entropy_with_logits(logits = self.scores, labels = self.y)
                  #损失函数
self.loss = tf.reduce_mean(self.losses) + self.lambda1 * self.l2_loss
tf.summary.scalar('loss', self.loss)
self.optimizer = tf.train.AdamOptimizer(self.learning_rate).minimize(self.losses)
                  #优化器
self.correct_predictions = tf.equal(self.predictions, tf.round(self.y))
self.accuracy = tf.reduce_mean(tf.cast(self.correct_predictions, "float"), name = "
accuracy")
tf.summary.scalar('accuracy', self.accuracy)
```

3. 模型实现

使用 tf.train.write_graph()函数将模型图定义保存到 model.pbtxt 文件中,训练完成后,使用 tf.train.Saver()函数将权重保存在 model_ckpt 中。model.pbtxt 和 model_ckpt 文件将被用于创建 protobuf 格式的 TensorFlow 模型优化版本,以便与 Android 应用集成,相关代码如下:

```
for epoch in range(self.epochs): #轮次
  gen_batch = self.batch_gen(self.X_train, self.y_train, self.batch_size)
  gen_batch_val = self.batch_gen(self.X_val,self.y_val,self.batch_size_val)
  for batch in range(self.num_batches): #批次
    X_batch, y_batch = next(gen_batch)
    X_batch_val, y_batch_val = next(gen_batch_val)
    sess.run(self.optimizer,feed_dict = {self.X: X_batch, self.y: y_batch})
    if (batch + 1) % 10 == 0:
      c, a = sess.run([self.loss, self.accuracy], feed_dict = {self.X: X_batch, self.y: y_batch})
      print(" Epoch = ", epoch + 1, " Batch = ", batch + 1, " Training Loss: ", "{:.9f}".format(c),
      " Training Accuracy = ", "{:.9f}".format(a))
#模型权值保存相关代码
builder = tf.saved_model.builder.SavedModelBuilder(saved_model_dir)
builder.add_meta_graph_and_variables(sess, [tf.saved_model.tag_constants.SERVING],
        signature_def_map = {
        tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY: signature},
        legacy_init_op = legacy_init_op)
builder.save()
tflite_model = tf.contrib.lite.toco_convert(sess.graph_def, [self.X[0]], [self.prediction_
probability[0]],inference_type = 1, input_format = 1, output_format = 2,quantized_input_stats
 = None, drop_control_dependency = True)
open(self.path + "converted_model.tflite", "wb").write(tflite_model)
```

在 train()函数中,根据传入批量大小使用生成器生成随机批次,生成器函数的定义如下:

```
def batch_gen(self, X, y, batch_size):
    index = list(range(X.shape[0]))
    np.random.shuffle(index)
    batches = int(X.shape[0] // batch_size)
    for b in range(batches):
        X_train,y_train = X[ index[b * batch_size: (b + 1) * batch_size], :], y[
            index[b * batch_size: (b + 1) * batch_size]]
        yield X_train, y_train
```

通过合适的参数调用函数,创建批量的迭代器对象。使用 next()函数,提取批量对象的下一个对象。在每个轮次开始时调用生成器函数,以保证每个轮次中的批量都是随机的。

3.3.3　模型保存

在 model.pbtxt 和 model_ckpt 的文件中保存训练好的模型并不能直接被 Android 应用程序使用。需要将其转换为 protobuf 格式(扩展名为.pb 文件),与 Android 应用集成。优化的 protobuf 格式小于 model.pbtxt 和 model_ckpt 文件的大小。

首先,定义输入张量和输出张量的名称;其次,通过 tensorflow.python.tools 中的 freeze_graph 函数,使用这些输入和输出张量以及 model.pbtxt 和 model_ckpt 文件,将模型冻结;最后,被冻结的模型通过 tensorflow.python.tools 中的 optimize_for_inference_lib

函数进一步优化,创建 protobuf 模型(即 optimized_model.pb),相关代码如下:

```
freeze_graph.freeze_graph(input_graph_path, input_saver_def_path,
                  input_binary, checkpoint_path, output_node_names,
                  restore_op_name, filename_tensor_name,
                  output_frozen_graph_name, clear_devices, "")
input_graph_def = tf.GraphDef()
with tf.gfile.Open(output_frozen_graph_name, "rb") as f:
    data = f.read()
    input_graph_def.ParseFromString(data)
output_graph_def = optimize_for_inference_lib.optimize_for_inference(
        input_graph_def,
        ["inputs/X"],                    #输入节点构成的数组
        ["positive_sentiment_probability"],
        tf.int32.as_datatype_enum        #输出节点构成的数组
        )
#保存优化后的模型图
f = tf.gfile.FastGFile(output_optimized_graph_name, "w")
f.write(output_graph_def.SerializeToString())
```

3.3.4 词典保存

在预处理期间,训练 Keras tokenizer,将单词替换为数字索引,处理后的电影评论提供给 LSTM 模型进行训练。保留频率最高的前 50 000 个单词,并将电影评论序列的最大长度限制为 1000。尽管训练后的 Keras tokenizer 被保存并用于推断,但不能直接被 Android 应用程序使用。

将 Keras tokenizer 还原,50 000 个单词及其相应的单词索引保存在文本文件中。此文本文件可以在 Android 应用程序中使用,以构建单词到索引的词典,用来转换电影评论的文本。单词到索引映射可以通过 tokenizer.word_index 从加载的 Keras tokenizer 对象进行检索。相关代码如下:

```
def tokenize(path,path_out):
    #保存词典
    with open(path, 'rb') as handle:
        tokenizer = pickle.load(handle)
    dict_ = tokenizer.word_index
    keys = list(dict_.keys())[:50000]
    values = list(dict_.values())[:50000]
    total_words = len(keys)
    f = open(path_out,'w')
    for i in range(total_words):
        line = str(keys[i]) + ',' + str(values[i]) + '\n'
        f.write(line)
    f.close()
```

3.3.5　模型测试

完成模型训练后，移植到移动端，在设计移动应用程序时包括交互界面设计及核心逻辑设计。

1. 交互界面设计

移动应用程序界面设计的相应代码采用 XML 文件格式。应用程序包含一个简单的电影评论文本框，用户在其中输入他们对于电影的评论，完成后单击 SUBMIT 按钮，电影评论将被传递给应用程序的核心逻辑模块，该模块处理电影评论文本，并将其传递给 TensorFlow 优化模型进行推断，针对电影评论的情感打分，该分数会转换为相应的星级，并显示在移动应用程序中。

用于帮助用户和移动应用程序核心逻辑进行彼此交互的变量是在 XML 文件中通过 android：id 选项声明的。例如，用户提供的电影评论可以使用 Review 变量进行处理，对应 XML 文件中的定义为：

```
android: id = "@ + id/submit"
```

相关代码如下：

```
res/layout/activity_main.xml
<?xml version = "1.0" encoding = "utf - 8"?>
< android.support.constraint.ConstraintLayout xmlns:android = " http://schemas.android.com/
apk/res/android"
  xmlns:app = "http://schemas.android.com/apk/res - auto"
  xmlns:tools = "http://schemas.android.com/tools"
  android:layout_width = "match_parent"
  android:layout_height = "match_parent"
  tools:context = ".MainActivity"
  tools:layout_editor_absoluteY = "81dp">
  < TextView
     android:id = "@ + id/desc"
     android:layout_width = "100dp"
     android:layout_height = "26dp"
     android:layout_marginEnd = "8dp"
     android:layout_marginLeft = "44dp"
     android:layout_marginRight = "8dp"
     android:layout_marginStart = "44dp"
     android:layout_marginTop = "36dp"
     android:text = "Movie Review"
     app:layout_constraintEnd_toEndOf = "parent"
     app:layout_constraintHorizontal_bias = "0.254"
     app:layout_constraintStart_toStartOf = "parent"
     app:layout_constraintTop_toTopOf = "parent"
     tools:ignore = "HardcodedText" />
```

```
< EditText
    android:id = "@ + id/Review"
    android:layout_width = "319dp"
    android:layout_height = "191dp"
    android:layout_marginEnd = "8dp"
    android:layout_marginLeft = "8dp"
    android:layout_marginRight = "8dp"
    android:layout_marginStart = "8dp"
    android:layout_marginTop = "24dp"
    app:layout_constraintEnd_toEndOf = "parent"
    app:layout_constraintStart_toStartOf = "parent"
    app:layout_constraintTop_toBottomOf = "@ + id/desc" />
< RatingBar
    android:id = "@ + id/ratingBar"
    android:layout_width = "240dp"
    android:layout_height = "49dp"
    android:layout_marginEnd = "8dp"
    android:layout_marginLeft = "52dp"
    android:layout_marginRight = "8dp"
    android:layout_marginStart = "52dp"
    android:layout_marginTop = "28dp"
    app:layout_constraintEnd_toEndOf = "parent"
    app:layout_constraintHorizontal_bias = "0.238"
    app:layout_constraintStart_toStartOf = "parent"
    app:layout_constraintTop_toBottomOf = "@ + id/score"
    tools:ignore = "MissingConstraints" />
< TextView
    android:id = "@ + id/score"
    android:layout_width = "125dp"
    android:layout_height = "39dp"
    android:layout_marginEnd = "8dp"
    android:layout_marginLeft = "96dp"
    android:layout_marginRight = "8dp"
    android:layout_marginStart = "96dp"
    android:layout_marginTop = "32dp"
    android:ems = "10"
    android:inputType = "numberDecimal"
    app:layout_constraintEnd_toEndOf = "parent"
    app:layout_constraintHorizontal_bias = "0.135"
    app:layout_constraintStart_toStartOf = "parent"
    app:layout_constraintTop_toBottomOf = "@ + id/submit" />
< Button
    android:id = "@ + id/submit"
    android:layout_width = "wrap_content"
    android:layout_height = "35dp"
    android:layout_marginEnd = "8dp"
    android:layout_marginLeft = "136dp"
```

```
    android:layout_marginRight = "8dp"
    android:layout_marginStart = "136dp"
    android:layout_marginTop = "24dp"
    android:text = "SUBMIT"
    app:layout_constraintEnd_toEndOf = "parent"
    app:layout_constraintHorizontal_bias = "0.0"
    app:layout_constraintStart_toStartOf = "parent"
    app:layout_constraintTop_toBottomOf = "@ + id/Review" />
</android.support.constraint.ConstraintLayout >
```

该文件提供了 5 个控件。其中：1 个 Button，用于提交电影评论；2 个 TextView，分别显示影评和预测电影评论为正面的概率；1 个 RatingBar，显示星级评分；1 个 EditText，获取用户输入的评论。

2. 核心逻辑设计

Android 应用程序的核心逻辑是处理用户请求以及传递的数据，将结果返回给用户。作为应用程序的一部分，核心逻辑将接收用户提供的电影评论，并处理原始数据，将其转换为可以被训练好的 LSTM 模型进行推断的格式。

Java 中的 OnClickListener()函数用于监视用户是否已提交处理请求。在可以将数据输入经过优化训练好的 LSTM 模型进行推断之前，用户提供电影评论中的每个单词都需要被转化为索引。因此，除了优化 protobuf 模型，单词字典及其对应的索引也需要预先存储在设备上。使用 TensorFlowInferenceInterface()方法通过训练好的模型来运行推断。经过优化的 protobuf 模型和单词字典及其相应的索引存储在 assets 文件夹中。

应用程序核心逻辑需要完成的任务如下：

(1) 将单词到索引的字典加载到 WordToInd HashMap 中。单词到索引字典是在训练模型之前预处理文本时从 tokenizer 派生而来的。相关代码如下：

```
final Map < String, Integer > WordToInd = new HashMap < String, Integer >();
        BufferedReader reader = null;
        try { //单词到索引的字典加载
          reader = new BufferedReader(
                new InputStreamReader(getAssets().open("word_ind.txt")));
          String line;
          while ((line = reader.readLine()) != null)
          {//读入
              String[] parts = line.split("\n")[0].split(",",2);
              if (parts.length >= 2)
              {
                  String key = parts[0];
                  int value = Integer.parseInt(parts[1]);
                  WordToInd.put(key,value);
              } else
              {
              }
```

```
            }
        } catch (IOException e) { //捕捉异常
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                }
            }
        }
```

(2) 通过监听 OnClickListener()方法判断用户是否已提交电影评论进行推断。

(3) 如果已提交,则从 XML 绑定的 Review 对象中读取。

首先,通过删除标点符号等操作清理评论文本; 其次,进行单词分词。每个单词都使用 WordToInd HashMap 转换为相应的索引。这些索引构成输入 TensorFlow 模型并用于推断的 InputVec 向量,向量的长度为 1000。因此,如果评论少于 1000 个单词,则用 0 在向量开头进行填充。相关代码如下:

```
final Map<String,Integer> WordToInd = new HashMap<String,Integer>();
        BufferedReader reader = null;
        try {  //读入缓存
            reader = new BufferedReader(
                new InputStreamReader(getAssets().open("word_ind.txt")));
            String line;
            while((line = reader.readLine()) != null)
            {
                String[] parts = line.split("\n")[0].split(",",2);
                if(parts.length >= 2)
                {
                    String key = parts[0];
                    int value = Integer.parseInt(parts[1]);
                    WordToInd.put(key,value);
                } else
                {
                }
            }
        } catch(IOException e) {  //捕捉异常
        } finally {
            if(reader != null) {
                try {
                    reader.close();
                } catch(IOException e) {
                }
            }
        }
```

(4) 从 assets 文件夹将经过优化的 protobuf 模型(扩展名为. pb)载入内存,使用 TensorFlowInferenceInterface 功能创建 mInferenceInterface 对象,与原始模型一样,需要定义输入/输出节点,相关代码如下:

```
private TensorFlowInferenceInterface mInferenceInterface;
private static final String MODEL_FILE = "file:///android_asset/optimized_model.pb";
//模型存放路径
private static final String INPUT_NODE = "inputs/X";
private static final String OUTPUT_NODE ="positive_sentiment_probability";
```

对于模型,它们被定义为 INPUT_NODE 和 OUTPUT_NODE,分别包含 TensorFlow 输入占位符的名称和输出的评分概率操作。mInferenceInterface 对象的 feed()方法用于将 InputVec 赋值给模型的 INPUT_NODE,而 mInferenceInterface 的 run()方法用于执行 OUTPUT_NODE。最后,调用 mInferenceInterface 的 fetch()得到用浮点变量 value_表示推断结果。相关代码如下:

```
mInferenceInterface.feed(INPUT_NODE,InputVec,1,1000);
mInferenceInterface.run(new String[] {OUTPUT_NODE}, false);
System.out.println(Float.toString(value_[0]));
mInferenceInterface.fetch(OUTPUT_NODE, value_);
  System.out.println(Float.toString(value_[0]));
```

(5) 首先,将 value_乘以 5 得到情感得分(评论为正面评论的概率);其次,提供给 Android 应用程序的交互对象 ratingBar 变量。相关代码如下:

```
  double scoreIn;
scoreIn = value_[0] * 5;
double ratingIn = scoreIn;
String stringDouble = Double.toString(scoreIn);
score.setText(stringDouble);
  ratingBar.setRating((float) ratingIn);
```

此外,还需要编辑应用程序的 build.gradle 文件,将需要的包添为依赖项。

3.4　系统测试

本部分包括数据处理、模型训练、词典保存及模型效果。

3.4.1　数据处理

在 PyCharm 终端输入 python preprocess.py --path E:/MRSA/aclImdb/,输出结果如图 3-8 所示。

```
(tensorflow) C:\Users\dy-d\PycharmProjects\MRSA>python preprocess.py --path E:/MRSA/aclImdb/
Using TensorFlow backend.
records_processed 50000
5.458 min: Process
```

图 3-8　数据处理输出结果

3.4.2　模型训练

在 PyCharm 终端输入如下命令：

```
python movie_review_model_train.py process_main -- path E:/MRSA/ -- epochs 10
```

开始训练，模型经过 10 个轮次的适度训练，避免过拟合。优化器的学习率为 0.001，训练和验证的批量大小分别设置为 250 和 50。将训练输出结果保存在.txt 文件中。BasicLSTM 的训练结果如图 3-9 所示，MultiLSTM 训练结果如图 3-10 所示。

```
250 50
 Epoch= 1  Validation Loss:  0.616187274  Validation Accuracy= 0.680000007
 Epoch= 2  Validation Loss:  0.524188161  Validation Accuracy= 0.740000010
 Epoch= 3  Validation Loss:  0.436133772  Validation Accuracy= 0.819999993
 Epoch= 4  Validation Loss:  0.390949339  Validation Accuracy= 0.819999993
 Epoch= 5  Validation Loss:  0.515198827  Validation Accuracy= 0.759999990
 Epoch= 6  Validation Loss:  0.343094289  Validation Accuracy= 0.860000014
 Epoch= 7  Validation Loss:  0.261696249  Validation Accuracy= 0.939999998
 Epoch= 8  Validation Loss:  0.244921491  Validation Accuracy= 0.899999976
 Epoch= 9  Validation Loss:  0.411403835  Validation Accuracy= 0.839999974
 Epoch= 10  Validation Loss:  0.355748057  Validation Accuracy= 0.879999995
 Test Loss:  0.295403659  Test Accuracy= 0.892000020
4.527 hrs: Model train
```

图 3-9　BasicLSTM 训练结果

```
250 50
 Epoch= 1  Validation Loss:  0.538058639  Validation Accuracy= 0.759999990
 Epoch= 2  Validation Loss:  0.555779696  Validation Accuracy= 0.759999990
 Epoch= 3  Validation Loss:  0.495759934  Validation Accuracy= 0.779999971
 Epoch= 4  Validation Loss:  0.456312269  Validation Accuracy= 0.839999974
 Epoch= 5  Validation Loss:  0.342410803  Validation Accuracy= 0.839999974
 Epoch= 6  Validation Loss:  0.361103296  Validation Accuracy= 0.860000014
 Epoch= 7  Validation Loss:  0.477218360  Validation Accuracy= 0.779999971
 Epoch= 8  Validation Loss:  0.324074388  Validation Accuracy= 0.839999974
 Epoch= 9  Validation Loss:  0.373119235  Validation Accuracy= 0.800000012
 Epoch= 10  Validation Loss:  0.378401011  Validation Accuracy= 0.839999974
 Test Loss:  0.362354994  Test Accuracy= 0.856000006
208912.500 s: Model train
```

图 3-10　MultiLSTM 训练结果

通过对比，MultiLSTM 模型训练集的准确率达到 94%，在验证集、测试集上的准确度均随着训练的进展而减少，发生了过拟合现象。所以最终移植到 Android 端时，采用 BasicLSTM 模型。

3.4.3　词典保存

在 Pycharm 终端输入 `python freeze_code.py -- path E:/MRSA/ -- MODEL_NAME`

model，将模型冻结为 protobuf 格式，终端输出运行时间为 1.177min。

在 Pycharm 终端输入 python tokenizer_2_txt.py --path 'E:/MRSA//aclImdb/tokenizer.pickle' --path_out 'E:/MRSA/word_ind.txt'，即可保存词典。

保存的 optimized_model.pb 和 word_ind.txt 文件会移植到移动端。

3.4.4　模型效果

本部分包括程序下载运行和应用使用说明。

1. 程序下载运行

Android 项目编译成功后，在真机上进行测试，模拟器运行较慢，不建议使用。运行到真机的方法如下：

(1) 将手机数据线连接到计算机，开启开发者模式，打开 USB 调试，单击 Android 项目的“运行”按钮，将出现连接手机选项，单击即可。

(2) Android Studio 生成.apk 文件，发送到手机，在手机上下载.apk 文件，安装即可。

2. 应用使用说明

打开 App，应用初始界面如图 3-11 所示。

界面从上至下分别是文本框显示 Movie Review、文本编辑框、按钮、文本框显示概率，RatingBar 显示评分值。

此时在文本框内输入有关电影 *The Shawshank Redemption* 的评论，如图 3-12 所示。

图 3-11　应用初始界面

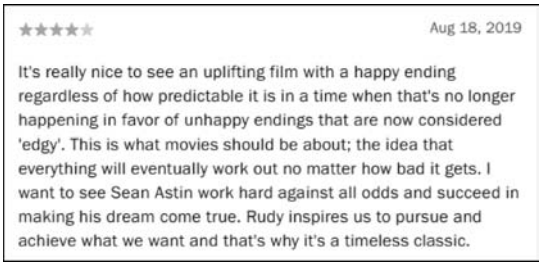


图 3-12　*The Shawshank Redemption* 电影评论

单击 SUBMIT 按钮,显示文本框内输出预测概率为 4.293/5,如图 3-13 所示。而评论员给这部电影的评分为 4/5,预测的评分更加精细化。

单击 SUBMIT 按钮,显示文本框内输出预测概率为 3.246/5,如图 3-14 所示。而 Rotten Tomatoes 对这部电影的平均评分为 3.5/5。

图 3-13　移动应用预测结果 1

图 3-14　移动应用预测结果 2

从对比结果可以看出,应用程序能够为电影评论提供更加精细化的评分,用于电影评分的修正。以上各电影评分来自如下两个链接:

(1) https://www.rottentomatoes.com/m/shawshank_redemption/reviews?type=user;

(2) https://www.rottentomatoes.com/m/interstellar_2014/。