Project 1: Spam Email Detection

一、数据处理及模型建立

1.Email Embedding

此处进行的预处理方式与上次作业的相似。所以只进行简单的描述。

进行处理的步骤如下：

分词，去除stop words，去除短的字符串，提取词干，去除数字。

然后对预处理后的email进行word count，分成ham和spam分别进行。我们对其中的词频进行排序，分别取ham、spam中出现频率最高的4000个词作为之后embedding的依据。因为二者之间存在大量重复，所以进行`set`操作后只剩下5522个词。

之后，我们就**用这5522个词对每一个文本进行清洗，只保留这5522个词，然后计数。**最后，每一个文本用一个长度为5522维的向量表示，其中`第i维`上的数值，表示了单词w_i在此文本出现的次数。

这样我们就生成了一个`29991x5522`维的稀疏矩阵来表达我们的训练数据。

其中前`14786`行是ham训练数据的embedding，后`15205`行是`spam训练数据的embedding。`

最后我们把这个`29991x5522`维的训练矩阵以`txt`文件的形式输出，供后续的用`hadoop`训练`Naive Bayes`时使用。

2.Hadoop训练Naive Bayes模型

首先是使用`Hadoop`进行每个类别各个单词出现的总频数和每个类别所有单词出现总频数的计算。这两组值分别记为N_{yi}、$N_y = \sum_{i=1}^{n} N_{yi}$

具体核心代码如下：

首先实现`mapper`，它将传给他的一个文章的`embedding array`进行`mapping`，它的输出是一个键值对，`<<Word_Id, Class_Id>, Wi>`，`Wi`表示序列为`Word_ID`单词在这个文档出现的次数，这个是从传入的`array`中直接得来的。

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.MapWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class NaiveBayesMapper extends
Mapper<IntWritable[],IntWritable,MapWritable,IntWritable> {
    IntWritable v = new IntWritable();
    MapWritable k = new MapWritable<IntWritable,IntWritable>()
    @Override
    protected  void map(IntWritable[] key, IntWritable classtype,Context
context){
        for (int i=0;i<key.length ;i++){
            k.set(i,classtype);
            v.set(key[i]);
            context.write(k,v);
```

```
        }

    }
}
```

然后实现`Reducer`，`Reducer`的输入是`<<Word_Id, Class_Id>, {count1, count2, ..., countn}>`。其中`{count1, count2, ..., countn}`是第1到n个样本中序列为`Word_ID`单词出现的次数。在 **Reduce**里把`count1，count2，…，countn`求和，就得到了序列为`Word_ID`单词在 **Class_Id** 的类里出现的总次数**TotalCount**，也就是N_{yi}.

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.io.MapWritable;

public class NaiveBayesReducer extends Reducer<MapWritable, IntWritable,
MapWritable, IntWritable>{
    @Override
    protected void reduce(MapWritable key, Iterable<IntWritable> values,
                          Context context) throws IOException,
InterruptedException {

        // 1 统计单词总个数
        int sum = 0;
        for (IntWritable count : values) {
            sum += count.get();
        }

        // 2 输出单词总个数
        context.write(key, new IntWritable(sum));
    }
}
```

下面是`Hadoop`主程序的入口：`NaiveBayesDriver.class`

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.io.MapWritable;

public class NaiveBayesDriver {
    public static void main(String[] paths) throws IOException,
ClassNotFoundException, InterruptedException{
        // 1 获取job信息
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 获取jar包位置
        job.setJarByClass(NaiveBayesDriver.class);

        // 3 关联自定义的mapper和reducer
        job.setMapperClass(NaiveBayesMapper.class);
```

```
        job.setReducerClass(NaiveBayesReducer.class);

        // 4 设置map输出数据类型
        job.setMapOutputKeyClass(MapWritable<IntWritable,IntWritable>.class);
        job.setMapOutputValueClass(IntWritable.class);

        // 5 设置最终输出数据类型
        job.setOutputKeyClass(MapWritable<IntWritable,IntWritable>.class);
        job.setOutputValueClass(IntWritable.class);

        // 6 设置输入和输出文件路径
        for(int i=0; i < paths.size(); i++) {
            FileInputFormat.addInputPath(job, paths.get(i));
        }
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 7 提交代码
        boolean result = job.waitForCompletion(true);
        System.exit(result?0:1);
    }
}
```

这样我们就通过`MapReduce`得到了每个类别各个单词出现的总频数N_{yi}

通过以下公式就可以很简单的求得相应的`likelihood`

$$\hat{P}(x_i|y) = \hat{\theta}_{yi} = \frac{N_{yi}+\alpha}{N_y+\alpha n}$$

$$\text{where } N_{yi} = \sum_{x \in T} x_i \text{ is the number of times feature } i \text{ appears in a sample of class } y \text{ in the training set } T, \text{ and } N_y = \sum_{i=1}^{n} N_{yi} \text{ is the total count of all features for class } y.$$

然后再乘上`prior`，就可以进行比较，然后预测相应的类别

因为，每个文本embedding的向量较长，为了防止计算值下溢，我们把**概率相乘变成对数概率相加**，可以防止数字过小造成的问题。

得到`likelihood`的代码如下：

```
public class LogLikelihood {
    public static  double[] get_log_likelihood(int[] N_yi, double alpha, int n){
        assert N_yi.length == n;
        int Ny = 0;
        for (int i=0; i<N_yi.length;i++){
            Ny += N_yi[i];
        }
        double[] log_likelihood = new double[n];
        for (int i=0; i<n;i++){
            log_likelihood[i] = Math.log((1.0 * N_yi[i] + alpha) / (1.0 * Ny +
alpha*n));

        }
        return log_likelihood;
    }
}
```

之后我们把得到的`log_likelihood`组合成一个`2x5522`的二维数组，然后输出成`txt`文件，供后续存放进`Hbase`使用。

3.Python实现Naive Bayes模型

除了进行Hadoop的训练以外，我们还另外实现了一套Python的Naive Bayes模型。

代码如下：

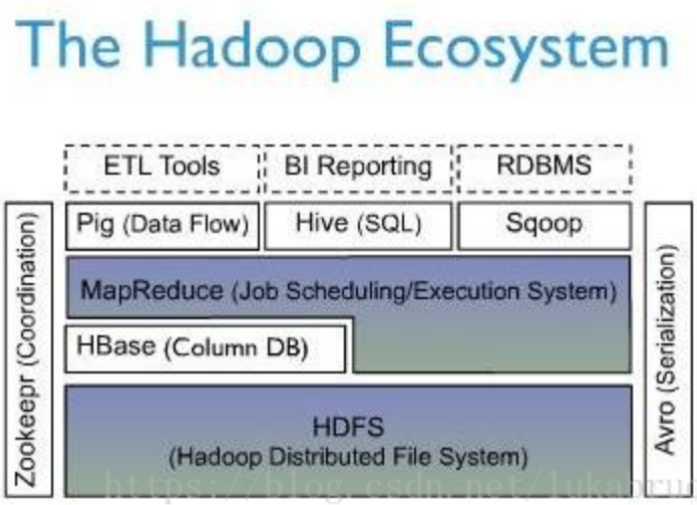
```
import numpy as np

class NaiveBayes(object):
    def __init__(self, alpha=1.0, fit_prior=True, class_prior=None):
        self.alpha = alpha
        self.fit_prior = fit_prior
        self.class_prior = class_prior

    def fit(self, X, y):
        self.class_type_, self.class_count_ = np.unique(y, return_counts=True)
        self.n_classes = len(self.class_type_)
        self.n_features = X.shape[1]
        if self.class_prior is None:
            if self.fit_prior:
                self.class_log_prior_ = np.log(self.class_count_ /
self.class_count_.sum())
        else:
            self.class_log_prior_ = np.log(self.class_prior)

        self.feature_count_ = np.zeros((self.n_classes, self.n_features))

        for i in range(len(self.class_type_)):
            self.feature_count_[i,:] = (X * (y ==
self.class_type_[i]).reshape(X.shape[0], 1)).sum(axis=0)

        self.Ny = self.feature_count_.sum(axis=1, keepdims=True)

        # shape (n_classes, n_features)
        self.feature_log_prob_ = np.log( (self.feature_count_ + self.alpha) /
(self.Ny + self.alpha*self.n_features) )


    def predict_log_proba(self, X):
        assert X.shape[1] == self.n_features, "X.shape[1] is not equal to
num_features"
        num_test = X.shape[0]
        log_prab = np.zeros((num_test, self.n_classes))
        for i in range(num_test):
            log_prab[i,:] = (X[i:i+1,:] * self.feature_log_prob_).sum(axis=1) +
self.class_log_prior_

        return log_prab


    def predict(self, X):
        log_prob = self.predict_log_proba(X)
        return np.argmax(log_prob, axis=1)
```

可以直接实现一整套的Naive Bayes的**训练和预测。**

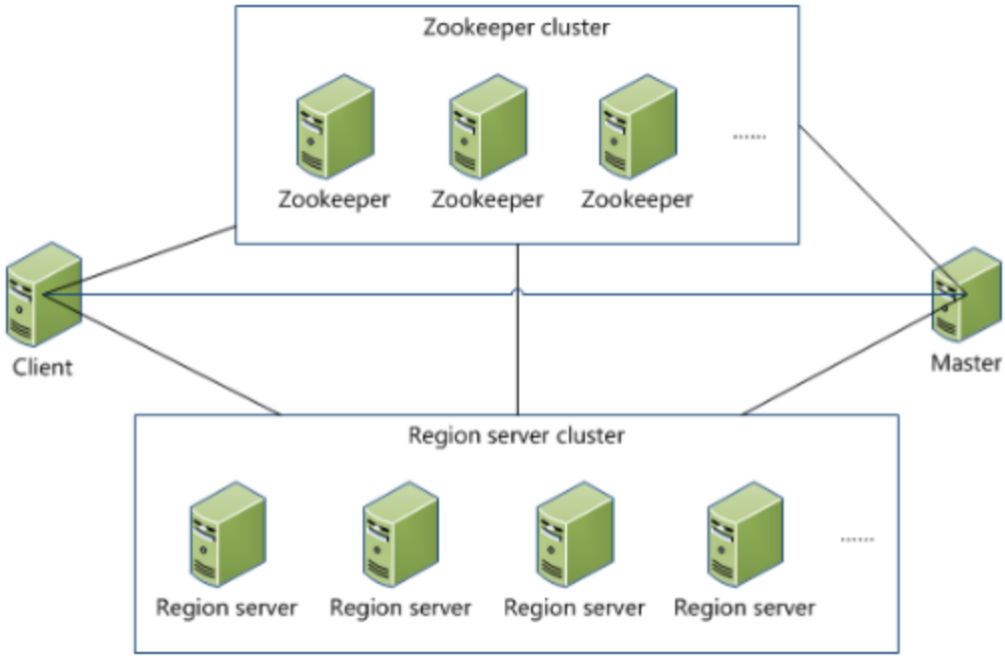
二、Hbase理解

Hbase是bigtable的开源山寨版，建立在hdfs之上。由于在实时性上，hdfs只是个文件系统，速度性能都十分受限，Hbase提供了高可靠性、高性能、列存储、可伸缩、实时读写的数据库系统。

Hbase是一个面向列的数据库，表可以设计的非常稀疏



Zookeeper是一种集群架构的维护服务，它保障了集群中master的产生以及实时监控Resion Server的状态，并将相关消息实时通知给Master。



三、Hbase 安装

1. Hbase下载

选择适合版本的下载镜像：https://www.apache.org/dyn/closer.lua/hbase/ 主要注意两点

- Java版本是否适合
- Hadoop版本是否适合

HBase Version	JDK 7	JDK 8	JDK 9 (Non-LTS)	JDK 10 (Non-LTS)	JDK 11
2.1+	✗	✓	! HBASE-20264	! HBASE-20264	! HBASE-21110
1.3+	✓	✓	! HBASE-20264	! HBASE-20264	! HBASE-21110

	HBase-1.3.x	HBase-1.4.x	HBase-1.5.x	HBase-2.1.x	HBase-2.2.x
Hadoop-2.4.x	✓	✗	✗	✗	✗
Hadoop-2.5.x	✓	✗	✗	✗	✗
Hadoop-2.6.0	✗	✗	✗	✗	✗
Hadoop-2.6.1+	✓	✗	✗	✗	✗
Hadoop-2.7.0	✗	✗	✗	✗	✗
Hadoop-2.7.1+	✓	✓	✗	✓	✗
Hadoop-2.8.[0-2]	✗	✗	✗	✗	✗
Hadoop-2.8.[3-4]	!	!	✗	✓	✗
Hadoop-2.8.5+	!	!	✓	✓	✓
Hadoop-2.9.[0-1]	✗	✗	✗	✗	✗
Hadoop-2.9.2+	!	!	✓	!	✓
Hadoop-3.0.[0-2]	✗	✗	✗	✗	✗
Hadoop-3.0.3+	✗	✗	✗	✓	✗
Hadoop-3.1.0	✗	✗	✗	✗	✗
Hadoop-3.1.1+	✗	✗	✗	✓	✓

由于我们的Hadoop版本是3.1.2，最终我们选择了Hbase2.2.1下载。

2. Hbase安装

方便起见，我们把Hbase装在了Hadoop目录下。

```
$ tar xzvf hbase-2.2.1-bin.tar.gz
$ cd hbase-2.2.1/
```

Hbase的配置文件主要在conf文件夹中，修改其中的hbase-env.sh、hbase-site.xml

1. bashrc

```
export PATH=$PATH:/usr/local/hadoop/hbase-2.2.1/bin
export HBASE_HOME=/usr/local/hadoop/hbase-2.2.1
export HBASE_CLASSPATH=$HBASE_HOME/conf
export HBASE_MANAGES_ZK=true
```

2. Hbase.env.sh

```
export JAVA_HOME=/usr/lib/jvm/default-java #配置java路径
export HBASE_CLASSPATH=/usr/local/hadoop/hbase-2.2.1/conf #配置Hbase的路径
export HBASE_MANAGES_ZK=true #使用自带的zookeeper，false的话需要自行配置zookeeper
```

3. hbase-site.xml

```
<configuration>
    <property>
    <name>hbase.rootdir</name>
    <value>file:///usr/local/hadoop/hbase-2.2.1/data</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>false</value>
  </property>
    <property>
    <name>hbase.unsafe.stream.capability.enforce</name>
    <value>false</value>
    <description>
      Controls whether HBase will check for stream capabilities (hflush/hsync).
      Disable this if you intend to run on LocalFileSystem, denoted by a rootdir
      with the 'file://' scheme, but be mindful of the NOTE below.
      WARNING: Setting this to false blinds you to potential data loss and
      inconsistent system state in the event of process and/or node failures. If
      HBase is complaining of an inability to use hsync or hflush it's most
      likely not a false positive.
    </description>
  </property>
    <property>
      <name>hbase.zookeeper.property.clientPort</name>
      <value>2181</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/usr/local/hadoop/hbase-2.2.1/data</value>
    </property>
    <property>
    <name>hbase.thrift.server.socket.read.timeout</name>
    <value>6000000</value>
    <description>eg:milisecond</description>
    </property>
</configuration>
```

大部分的错误都是与hbase-site.xml配置相关，有几点需要注意：

- 单机版与集群版：单机的rootdir就写file://本机路径，集群则hdfs://namenode:8020/hbase
- 集群需要加上

```
<property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
</property>
<property>
  <name>hbase.zookeeper.quorum</name>
  <value>node-a,node-b,node-c,...</value> #增加相关集群的节点
</property>
```

######4. Regionservers

加上hadoop2、hadoop3、hadoop4、hadoop5

3. Hbase测试

1. 启动

打开终端，使用start-hbase.sh启动hbase，如果是集群式的，使用jps可以看到下图所示的进程

```
root@hadoop1:~# jps
13809 SecondaryNameNode
14759 HQuorumPeer
15050 Jps
13067 ThriftServer
14076 ResourceManager
13452 NameNode
14895 HMaster
```

如果是单机版的，则只会看到一个HMaster进程

```
root@hadoop1:/usr/local/hadoop/hbase-2.2.1/conf# jps
13809 SecondaryNameNode
33811 Jps
33672 Main
30539 HMaster
13067 ThriftServer
14076 ResourceManager
13452 NameNode
```

输入hbase version，可以正常看到Hbase 2.2.1（对应的版本号）即暂时证明启动成功，因为后续我们还遇到了很多问题。

2.shell 测试

输入命令hbase shell可以进入shell，会有一点点慢，成功界面如下图所示

```
root@hadoop1:/usr/local/hadoop/hbase-2.2.1/conf# hbase shell
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.2.1, rf93aaf770cce81caacbf22174dfee2860dbb4810, 2019年 09月 10日 星期二 14:28:27 CST
Took 0.0042 seconds
hbase(main):001:0>
```

使用list指令查看已有的table，可以看到我们有一个叫PTable的表

```
hbase(main):001:0> list
TABLE
PTable
1 row(s)
Took 0.6390 seconds
=> ["PTable"]
```

使用scan可以简单看一看表中的数据，现在表是空的，还没有数据

```
hbase(main):003:0> scan "PTable"
ROW                                                COLUMN+CELL
0 row(s)
Took 0.1774 seconds
```

测试到现在可以证明Hbase已经正常安装且能够运行

4. 遇到的问题

1. Hbase启动后很快就退出了，Hmaster进程消失

这个造成的原因可能有很多，Zookeeper没有正常启动，hbase-site.xml没有配置好都会使得它发生错误。最好的解决方法就是到它的log文件中去查看log，它会显示相应的报错信息。

我们这里给出几个我们会遇到的问题：

```
Opening socket connection to server localhost/127.0.0.1:2181. Will not attempt
to authenticate using SASL (unknown error)
```

1. 单机情况下不需要quorum，这个property最好删掉，不写的时候它默认就是localhost
2. 如果用Python进行，需要修改hbase.thrift.server.socket.read.timeout，将它调大一些
3. 用对应端口访问时，检查下服务器的防火墙是否开启，是否可以正常访问开放的端口

2. Hbase启动时，有jar包的冲突

出现Class Path contains multiple SLF4J bindings，原因在于hadoop和hbase的lib中可能有相同的jar包。

1. 删掉hbase中的对应的jar包即可
2. 将hadoop的safemode关闭也可

```
hdfs dfsadmin -safemode leave
```

3. Hbase shell交互感觉很卡，发生error

这个错误其实是配置的问题，虽然你可以成功进入Hbase，但是由于配置的问题，会使得无法访问Hbase，因此输入相应指令时，Hbase会很卡，发生Error，此时你需要重新检查下hbase-site.xml的配置问题。

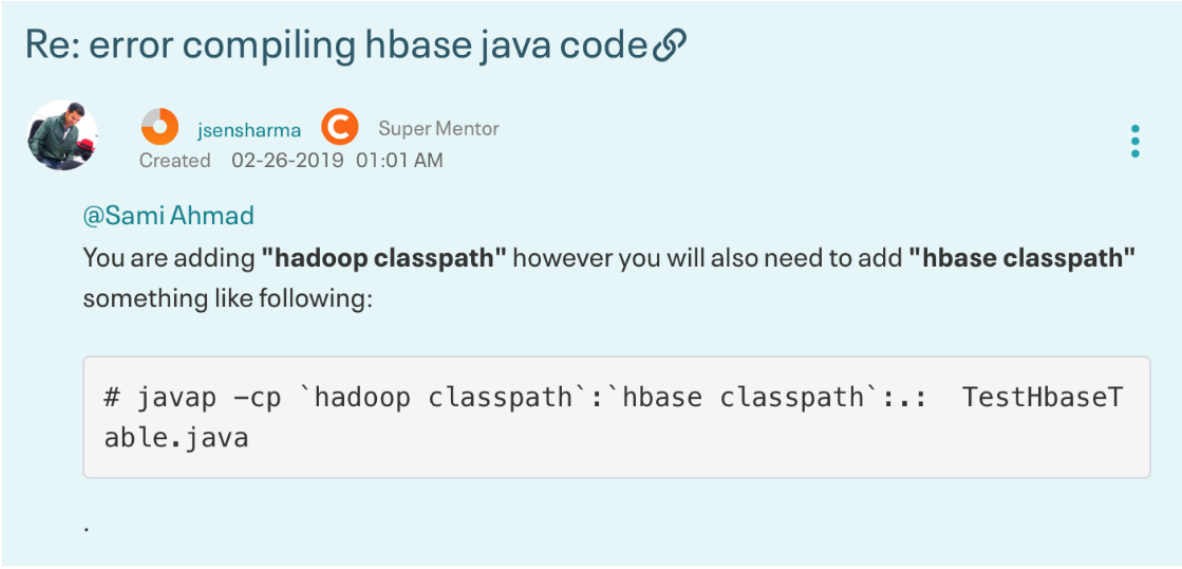
5. 参考文献

1. 官网：https://hbase.apache.org/book.html#faq
2. Hbase基本操作：https://www.cnblogs.com/xinfang520/p/7717399.html
3. Python配置：https://blog.csdn.net/luanpeng825485697/article/details/81048468

四、Java数据导入Hbase

1. java编译、打包

最常见的问题就是发现在使用javac编译时，可能由于路径的原因导致其无法定位到hbase相关的包，爆出x.x.hbase does not exist的错误。这个问题卡了我们很久，因为感觉环境变量已经进行了相关的配置，但它还是没办法定位到相关的包，最终在stack overflow上找到了可行的解决方案。



Re: error compiling hbase java code

jsensharma Super Mentor
Created 02-26-2019 01:01 AM

@Sami Ahmad

You are adding **"hadoop classpath"** however you will also need to add **"hbase classpath"** something like following:

```
# javap -cp `hadoop classpath`:`hbase classpath`:.:  TestHbaseTable.java
```

.

图中的javap变成javac，对应的java名称换掉，其余保持不变即可。

```
javac -cp `hadoop classpath`:`hbase classpath`:.: x.java
java -cp `hadoop classpath`:`hbase classpath`:.: x
```

2. Java创建表并插入数据到Hbase中

在hbase shell中可以通过create，put等命令创建数据表，并插入数据，但因为我们要处理大量的数据，更优雅的实现是通过Hbase的Java API实现。

1. 创建数据表

我们要新建一张名为“TABLE_NAME”的表，其中的列族名为“COLUMN_FAMILY”。

```
public static void createOrOverwrite(Admin admin, HTableDescriptor table) throws IOException {
        if (admin.tableExists(table.getTableName())) {
            admin.disableTable(table.getTableName());
            admin.deleteTable(table.getTableName());
            System.out.println("Rewrite");
        }
        admin.createTable(table);
    }

    public static void CreateTable(Configuration config) throws IOException{
        try (Connection connection = ConnectionFactory.createConnection(config);
        Admin admin = connection.getAdmin()) {
            HTableDescriptor table = new HTableDescriptor(TableName.valueOf(TABLE_NAME));
            table.addFamily(new HColumnDescriptor(COLUMN_FAMILY));

            System.out.print("Creating table. ");
            createOrOverwrite(admin, table);
            System.out.println("Table created. ");
        }
   }
```

为了避免与已经存在的表冲突，还写了createOrOverwrite函数。

在Hbase shell中用describe查看已建立的表的信息

```
hbase(main):002:0> describe 'PTable'
Table PTable is ENABLED
PTable
COLUMN FAMILIES DESCRIPTION
{NAME => 'P_ham', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION
_BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'fal
se', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLI
CATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_M
EMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN => '
false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}

{NAME => 'P_spam', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSIO
N_BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'fa
lse', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPL
ICATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_
MEMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN =>
'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}

{NAME => 'Word', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_
BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'fals
e', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLIC
ATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_ME
MORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN => 'f
alse', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}

3 row(s)

QUOTAS
0 row(s)
Took 0.3116 seconds
```

2.读取数据并插入

```
public static void InsertData(Configuration config) throws IOException {
    try (Connection connection = ConnectionFactory.createConnection(config);
         Table hTable = connection.getTable(TableName.valueOf(TABLE_NAME))) {
        List<String> list = readTxtFile(INPUT_PATH);
        for(int i = 0;i<list.size();i++){
            String line = list.get(i);
            String[] myArray = line.replaceAll("\\s+", "@").split("@");
            Put p = new Put(("row"+i).getBytes());

 p.addColumn(COLUMN_FAMILY.getBytes(),"ham".getBytes(),myArray[0].getBytes());

 p.addColumn(COLUMN_FAMILY.getBytes(),"spam".getBytes(),myArray[1].getBytes());
            hTable.put(p);
            hTable.close();
        }
    }
}
```

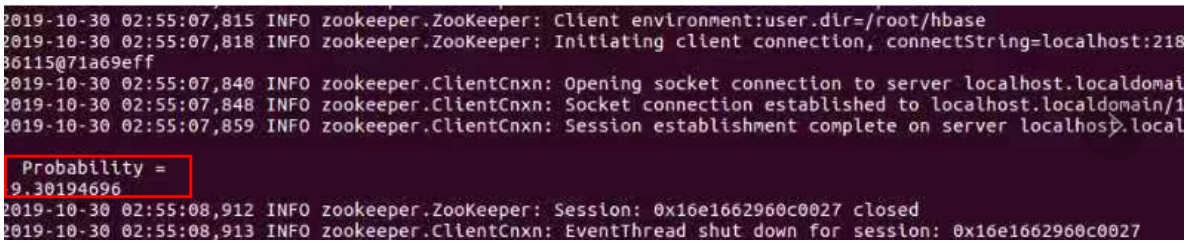
根据已有的文件路径INPUT_PATH读取数据，随后逐行存入到Hbase中。

在Hbase中使用scan显示表的数据，得到以下结果，可以看到分别有ham和spam的概率

```
row99                                    column=Probability:ham, timestamp=1572373257293, value=-10.84973652
row99                                    column=Probability:spam, timestamp=1572373257293, value=-7.74301710
row990                                   column=Probability:ham, timestamp=1572373258224, value=-5.09091362
row990                                   column=Probability:spam, timestamp=1572373258224, value=-4.75968478
row991                                   column=Probability:ham, timestamp=1572373258225, value=-13.15232161
row991                                   column=Probability:spam, timestamp=1572373258225, value=-10.17443507
row992                                   column=Probability:ham, timestamp=1572373258225, value=-10.09596472
row992                                   column=Probability:spam, timestamp=1572373258225, value=-8.93192860
row993                                   column=Probability:ham, timestamp=1572373258226, value=-8.51275000
row993                                   column=Probability:spam, timestamp=1572373258226, value=-8.00334263
row994                                   column=Probability:ham, timestamp=1572373258226, value=-9.43267050
row994                                   column=Probability:spam, timestamp=1572373258226, value=-9.57481395
row995                                   column=Probability:ham, timestamp=1572373258227, value=-14.53861598
row995                                   column=Probability:spam, timestamp=1572373258227, value=-9.60466691
row996                                   column=Probability:ham, timestamp=1572373258227, value=-11.24277911
row996                                   column=Probability:spam, timestamp=1572373258227, value=-9.71115039
row997                                   column=Probability:ham, timestamp=1572373258228, value=-10.98326791
row997                                   column=Probability:spam, timestamp=1572373258228, value=-9.98027905
row998                                   column=Probability:ham, timestamp=1572373258229, value=-8.94019402
row998                                   column=Probability:spam, timestamp=1572373258229, value=-7.63734266
row999                                   column=Probability:ham, timestamp=1572373258229, value=-9.61136229
row999                                   column=Probability:spam, timestamp=1572373258229, value=-9.29451198
5522 row(s)
Took 4.1246 seconds
```

3.读取表中数据

```
public static void ReadData(Configuration config,String rowKey,String col)
throws IOException {
        try (Connection connection = ConnectionFactory.createConnection(config);
        Table hTable = connection.getTable(TableName.valueOf(TABLE_NAME))) {
            Get get = new Get(rowKey.getBytes());
            get.addColumn(COLUMN_FAMILY.getBytes(),col.getBytes());
            Result result = hTable.get(get);
            System.out.println("\n  Probability =");
            System.out.println(new
String(result.getValue(COLUMN_FAMILY.getBytes(),col==null?
null:col.getBytes())));
            hTable.close();
        }
    }
```



以上是查询row88 spam的先验概率得到的输出。

3.遇到的问题

1.大多数博客上参考的代码编译失败

原因是最新版本的Hbase Java API启用了HTable方法，需要用Connection类来实例化，随后才可进行插入、查询数据等。

手动构建 HTable 已被**弃用**。请使用 **连接** 来实例化 **表**。

通过**连接**，可以使用 Connection.getTable（TableName）

<例如：

```
 Connection connection = ConnectionFactory.createConnection (config) ;

Table table = connection.getTable (TableName.valueOf (table1") ) ;

尝试
{
//根据需要使用表，对于单个操作和单个线程
}
finally
{
table.close () ;
connection.close () ;
}
```

五、算法实现

我们的任务是利用朴素贝叶斯算法进行正常、垃圾邮件的分类，这是一个有监督学习的二分类问题。我们首先通过Python对邮件文本进行预处理，然后利用Hadoop和Hbase来实现我们的算法。

1. 算法原理

本质上就是利用贝叶斯公式：

$$P(y|x_1,\dots,x_n)=\frac{P(y)P(x_1,\dots,x_n|y)}{P(x_1,\dots x_n)}$$

朴素贝叶斯时假设其特征彼此独立，因此可化简为：

$$y^*=\arg\min_y P(y)\prod_{i=0}^{n}P(x_i|y)$$

对于我们这个问题，我们只需计算其邮件种类的**先验概率**及每个特征在某种类下出现的**条件概率**即可。

$$\mathrm{先验概率}=\mathrm{每类邮件在总邮件中出现的概率}$$

$$\mathrm{条件概率}=\frac{N_{yi}+\alpha}{N_y+\alpha n}$$

我们使用

$$\alpha=1$$

即拉普拉斯平滑。

2. 算法框架实现

- 模型训练

1. Python对原始邮件数据进行预处理，构建邮件向量矩阵输出txt
2. 使用Hadoop的Mapreduce进行先验概率和条件概率的计算
3. 使用Java API将计算好的概率存入Hbase

- 模型测试

1. Python对测试数据进行预处理输出txt
2. 使用Hbase进行查询和预测

3. 朴素贝叶斯Java预测

我们已经把条件概率存在了Hbase中，使用的时候在ReadData函数中查询得到相对应的单词的条件概率。对于一批已经预处理的邮件，得到一个输入矩阵input，每一行表示一封邮件，每一列是一个单词对应的feature。我们对概率预先取了对数，因此在此只需要对应概率相加，并加上先验概率PRIOR_HAM和PRIOR_SPAM，比较两者的大小，即可以完成邮件的分类。我们将所有矩阵的分类结果存在数组result中。

```
public static int[] Prediction(Configuration config, double input[][]){
int result[] = new int[input.size()];
    for(int i=0;i<input.size();i++){            //traverse all the emails
        double p_ham, p_spam;
        p_ham = p_spam = 0;
        for(int j=0;j<input[0].size();j++){     //traverse all the features
            p_ham += input[i][j] * ReadData(config,"row"+i,"ham");
            p_spam += input[i][j] * ReadData(config,"row"+i,"spam");
```

```
            }
            p_ham += PRIOR_HAM;
            p_spam += PRIOR_SPAM;
            result[i] = (p_ham > p_spam);
        }
        return result;
    }
```

六、组员分工

实验部分

胡晨旭：数据处理及模型建立，算法尝试

王宇琪：Hbase安装，Hbase环境配置，模型测试

张智为：Hbase Java API测试，模型测试

报告部分

胡晨旭：数据处理及模型建立

王宇琪：Hbase部分及原理部分

张智为：报告整合