集成学习实验报告

清华大学计算机系 52 班 魏钧宇 2015011263

目录

1. 实验目的

2. 实验原理概述

1）集成学习的主要思想

2）Bagging 算法

3）AdaBoost 算法

3. 实验实现过程：

1）预处理：

<1> 数据整理与 Embedding

<2> 特征抽取

2）单一弱学习器的实现：

<1> SVM

<2> 决策树

<3> K 近邻

3）Bagging 集成方法的实现

4）AdaBoost 集成方法的实现：

<1> Weighted 方法

<2> EqualSample 方法

<3> ExtendSample 方法

4. 实验结果与分析：

1）各类模型在 Kaggle 平台上的测试结果：

<1> 单一学习器的效果

<2> Bagging 方法的实验结果

<3> Boosting 方法的实验结果

2）三种单一学习器的比较

3）Bagging 对三类学习器的影响分析

<1> SVM

<2> DTree

<3> KNN

4）Boosting 对两类学习器的影响分析

<1> SVM

<2> DTree

5. 总结与收获

1. 实验目的

1. 深入理解集成学习的主要思想，较为清楚 Bagging 和 Boosting 算法的实现细节

2. 学会使用 sklearn 库提供的各个弱学习器的函数库

3. 掌握特征选择及数据整理的基本方法

4. 提升 python 编程能力

2. 实验原理概述

1）集成学习的主要思想：

集成学习是通过一定的统计学手段，提升弱学习器的学习效果的一系列方法，粗略来看它可以以分成两类：一类是对于学习器本身的集成，一类是对于训练数据集的集成。

第一类基本思想是利用给定的多个学习器共同来预测出一个结果，典型的方法如 WeightedMajority 等。在本次实验中，我们主要关注的是第二类方法，即学习器只有一种，但是可以通过深度挖掘训练集的内在规律来提升学习器的效能。

2）Bagging 算法

Bagging 也叫拔靴法采样，它通过将训练数据集进行 N 次 Bootstrap 采样得到 N 个训练数据子集，对于每一个子集使用相同的算法分别训练得到一个模型，最终的结果将由这样 N 个模型通过投票得到。

Bagging 算法能够使用同一个数据集训练出不同的分类器，它的特点在于不同的分类器其实关注的是数据集的不同部分，从理论上来说同一个数据集的不同部分的重要性是一样的，所以最终得到的 N 个模型在投票时也不会有权重之分。

Bagging 算法还有一大好处在于每一个学习器的学习是并行完成的，前一个学习器的学习结果不会影响到后面学习的学习，具有极好的可并行性。

3）Boosting 算法

Boosting 算法的基本思想在于从错误中学习，前一个分类器在训练集上训练得到一个模型，在原有的训练集上进行测试会得到在这个训练集上的正确率，后一个模型就将主要的精力集中在前一个模型做错的例子上，进行带权重的训练。

Boosting 和 Bagging 一样也能够使用同一个数据集训练处不同的分类器，处于训练后期的分类器能够关注前一次分类器忽视的数据集部分，但是这样的关注所带来的效果是不同的，有一些学习器关注的了绝大多数的给定数据，有一些学习器则只关注了很少部分的数据（因为前一个学习器训练出的结果在绝大多数的例子上都是能够做对的，所以后面的学习器所关注的数据就不会有之前多，因此最终他们的投票权重也是不同的，这是 Boosting 和 Bagging 的一个很大的区别。

Boosting 在训练时的特点在于无法完成并行训练，因为后面分类器所基于的 Boosting 权重来自于之前的分类器的训练成果

3. 实验实现

1）预处理：

给定的训练集对应的 csv 文件有两个关键的维度：Review 和 Label，其中的 Review 是完成的分词的商品评论，也是我们所有特征的来源，Label 是这条评论对应的标签（即究竟是好评、中评还是差评），是本次试验的监督信号。训练集的总大小是 40000 余条。

针对这样的训练集，我们需要首先做一些简单的数据整理，然后在处理好的数据上做特征抽取，下面我们分别就这两点来介绍。

<1> 数据整理与评论的 embedding：

我们首先建立了评论中出现的所有词构成的词典（在 dic.txt 文件中），每一个词/符号都对应于特定的 id，基于这样的词典，我们就能够将给定的所有的评论表示成向量的形式，即在某一个维度 n 上的数值为 w 就表明 id 为 n 的词/符号对应在这条评论中出现了 w 次。如果不做特征的选择，我们的总特征有 11653 个。

<2> 特征抽取：

可以看到，上述的向量表示方式得到的评论对应的向量还是很稀疏的，有很多词或许在所有的评论中只出现了 1，2 次还需要专门使用一个维度来表示它，另一方面有一些词/符号对于我们的分类是没有意义的（比如逗号，‘你’，‘我’等）我们应当着重将这些词剔除出去，只保留真正有用的词作为我们的特征。

我尝试了两个方法来提取特征：基于 Gini 不纯度的计算法，基于互信息的计算法。

Gini 不纯度是用于衡量信息增益比的一个参数，它代表的模型的不纯度，如果基尼系数越小，则不纯度越低，特征越好。这和信息增益比恰好是相反的，Gini 系数的计算方法如下（假设有 k 个类别，第 k 个类别的概率是p_k，则基尼系数的表达式如下）

$$Gini(p) = \sum_{k=1}^{K} p_k * (1 - p_k)$$

在决策树的 C4.5 以及 ID3 算法中，这个方法被经常使用用于判断在分类时采用哪一个特征作为标准。在 sklearn 的 DTree 实现中有一个 featue_importance 的选项可以用于得到每一个特征的重要性，而这个重要性的计算就是基于 Gini 系数的，我选择了所有高于平均值的特征得到了新的字典 TreeDic.txt，这个字典将原来的 10000 多个下降到了 1163 个，足足下降了 10 倍对于提升训练的速度有很大的帮助。

除了使用 Gini 不纯度进行降维之外，我还尝试了基于互信息的降维方法，两个随机变量（在我们这里就是 Review 和 Label）的互信息定义为下：

$$I(X;Y) = \sum_{x\in X}\sum_{y\in Y} p(x,y) * \log(\frac{p(x,y)}{p(x) * p(y)})$$

在 feature.py 中我们自己实现了基于互信息的降维方法，得到了字典 my_dic.txt，他将原来的 10000 多个维度降低到了 2000 个。

那么究竟特称抽取的效果如何呢？我们在自己划分的验证集上（用 40000 做训练，2000 做测试）得到了如下的结果（用正确率表示）

	原始（1w+）	Gini（1163）	互信息（2000）
DTree	78.39%	77.84%	74.33%
SVM	83.8%	84.42%	80.29%

可以看出基于 Gini 系数的降维方法还是行之有效的，不仅没有带来太严重的正确率损失同时同时还极大的提升了训练的速度，而互信息的方法则带来的损失比较严重，所以在我们后面的实验中基本不采用这个方法。

2）单一弱学习器的实现

我在实验中一共尝试了 3 种弱学习器，他们的具体实现都是基于 sklearn 中的现成库的，同时 sklearn 还为我们提供了一些调参的接口，使用这些接口，我们能够对这些学习器做出一定的限制。所有单一弱学习器的实现均可参见文件 Signal.py

<1> DTree：

在 sklearn 给出的决策树实现中，可以通过制定 max_depth 来限定它的深度，同时在使用 fit 函数进行训练时还能够通过 sample_weight 来指定训练过程中的数据点的权重值。这个特性在后面实现 Boosting 中起到了较为关键的作用。

<2> SVM：

我试用了 sklearn 中的 LinearSVC，它的主要特点是训练要比普通的 SVC 快很多，但是暂时不能够提供 sample_weight 的功能。所以针对 SVM 的 Boosting 实现需要我们自己通过抽样的方法来完成，在后面介绍 Boosting 时，我会详细地介绍两种 boosting 的实现方法。

<3> KNN：

KNN 属于 Optional 的部分，所以我没有在它上花太多的精力进行参数调优，它的基本是现实还是基于 sklearn 库的，可以通过 n_neightbors 来指定第几近邻。

由于 KNN 对于特征的好坏要求非常大，我们的特征提取方法由相对比较原始，所以它的效果并不太好，在后面的实验结果部分可以明显看出这一点。

3）Bagging 方法的实现：

Bagging 的实现相对比较简单，在 ensemble.py 文件中的 Bagging 函数中完成。首先通过 BaggingSet 函数来生成 Bagging 集合，每次通过随机产生一个 0 – num – 1 的随机数来确定本轮中添加到新集合中的元素具体是哪个。得到 Bagging 集合以后，我们就可以使用分类器在这一系列的集合上进行训练了。

我还尝试了基于概率的预测方法，即使用 sklearn 提供的 predict_proba 预测在每一个维度上的概率值然后将所有 Bagging 的概率值取平均，最终得到的结果将不是一个（-1，0，1）的取值而是一个在他们中间的概率值，但是这个方法的效果并不好，所以在后面的实验中我们还是采用常规的预测方法。

4）AdaBoost 方法的实现：

由于这是一个多分类问题，所以我们实现的 AdaBoost 其实是 AdaBoost.M1 算法，完全按照课件上的要求来实现 AdaBoost.M1 方法，在 ensemble.py 的 AdaBoost 函数中可以看到我的实现，在此就不再赘述了，这里需要重点讨论的是 BoostSet 的生成，我一共尝试了三种方法，分别是 Weighted, EqualSample, ExtendSample

<1> Weighted 方法

Weighted 方法其实并没有额外的生成 Boosting 集合，在训练时依旧采用原始集合不变，在底层的弱分类器分类时会考虑到原始集合上不同样本的权重，然后根据这些权重训练获得不一样的结果。这个方式是最自然的 Boosting 实现，也是最能够体现 Boosting 的算法思想的方法，在决策树的 Boosting 学习中，我们使用的就是这样的方法，每一次决策树计数时会对于权重不同的样本给予特殊的考虑，遗憾的是在 LinearSVC 中没有提供 Sample_weighted 的接口，所以我们只能退而求其次，选用另外两种方法。

<2> EqaulSample 方法

EqualSample 方法实现的代码片段如下：

//首先将 Weights 进行归一化（如果所有样本等权重，则他们的权重为 1 / n，其中 n 为样本个数）

```
Weights= unify(Weights)
```

//接下来生成一个和之前的集合大小差不多的集合（这里的差不多是指新集合肯定比原集合更大，但是大小不会超过原集合的 2 倍）

```
while len(NI) < max_size: //此时的 max_size 为原集合大小（这也是将其称之为 EqualSampel 的原因）
    for i in range(0, len(Weights)):
        //下面的 size 也是原集合大小。
        // 生成一个从 0 到 size 的随机数，这样做的原因是在所有样本等权重时，他们刚好都能够被抽到，同时利用了 AdaBoost 中每一次只会降低正确样本的权重的做法，让正确的样本被抽中的概率更小，错误的样本被抽中的的概率更大。同时还能够保证权重高于均值的样本一定能够被抽到，低于均值的样本一定的概率被抽到。
        temp = random.uniform(0, 1.0 / size)
        if (temp < Weights[i]):
            NI.append(i) //加入到 NI 中表示被抽中
random.shuffle(NI)
```

这个就是 EqualSample 的实现方法，这个方法的好处在于能够保证新生成的抽样集合不会太大，同时很好的兼顾了各个不同样本的权重，但是这个方法的坏处在于不能够保证每一个 Sample 都被采样一次，这样就会导致一些训练信息的损失，这和实际的 Boosting 算法的思想是有一定的出入的。

<3> ExtendSample 方法

ExtendSample 方法的实现如下：

//首先将 Weights 进行归一化（如果所有样本等权重，则他们的权重为 1 / n，其中 n 为样本个数）

```
Weights= unify(Weights)
```

//这里先保证每一个元素至少都添加一次

```
for i in range(0, len(Reviews)):
    NI.append(i)
```

//接下来生成一个和之前集合 n 倍大小的集合差不多大小的集合（这里的 n 是一个可调参数成为 extend_time）

```
while len(NI) < max_size: //此时的 max_size 为新集合大小（这也是将其称之为 EqualSampel 的原因）
    for i in range(0, len(Weights)):
      // 生成一个从 0 到 size（这个 size 依旧是原集合大小）的随机数
       temp = random.uniform(0, 1.0 / size)
      if (temp < Weights[i]):
            NI.append(i) //加入到 NI 中表示被抽中
random.shuffle(NI)
```

ExtendSample 严格的按照了原理中 Boosting 算法的实现进行，即保证了信息不丢失，同时还兼顾了数据的权重，但是他的问题在于生成的集合太大，会影响训练速度。

在实验中结果中，我们将给出这两种方法在不同的分类器上的结果，同时进行分析。

4. 实验结果

1）各类集成方法在 Kaggle 上最好效果汇总

我在 Kaggle 平台上进行了大量的实验，尝试了几乎所有可能的模型，下面给出的是各类组合在 Kaggle 平台上得到的最好结果。

首先给出单一学习器的 RMSE 效果（均适用基于 Gini 方法抽取获得特征）便于和后面的实验做比对。

	Kaggle public board RMSE
SVM	0.33100
DTree	0.27063
KNN	0.54683

可以看出 KNN 的效果最差，DTree 的效果最好。

首先尝试 Bagging 方法的效果，这里均给出的是最好结果（包括 Kaggle public board RMSE 以及他们和单一学习器相比的提升或者下降，+号表示效果有所上升，-号表示结果有所下降）

	RMSE	Rate
SVM+Bagging	0.33102	不变
DTree+Bagging	0.24106	+10.93%
KNN+Bagging	0.51314	+6.16%

可以见得 Baggin 方法对于 SVM 的提升非常小，但是在 DTree 以及 KNN 中有很大的提升。

接下来展示的是 Boosting 方法的效果，和 Bagging 一样，这里同样给出的是最好的效果

	RMSE	Rate
SVM+Boosting(EqualSize)	0.31218	+5.69%
SVM+Boosting(ExtendSize)	0.35312	-6.68%
DTree+Boosting	0.26269	+2.93%

2）三种弱学习器的比较

在上一个段中我们给出了三种弱学期上的训练结果比较，我们发现决策树最好，KNN 最差，除此以外，我们还对比了他们的一些其他的性质。

在实验中我们发现，SVM 和 DTree 的训练时间都主要集中在 fit 函数（训练阶段），KNN 的训练时间主要集中在 predict 函数（预测阶段）从他们的实现过程分析可知，KNN 属于懒惰学习，它在 fit 阶段仅仅是通过 kd 树的形式记录下了每一个训练样本点的位置，在 predict 阶段才根据过去记录的样本点来判断当前待预测点的属性。相应的 SVM 和 DTree 在训练时已经建立好整个预测模型，新来的样本点只需要很少的时间就能够得到他们的答案。

此外我们还分析了 SVM 和 DTree 对于特征的敏感度，在特征选择时我们讨论过在特征的维度不同时，二者的表现情况：

	原始（1w+）	Gini（1163）	互信息（2000）
DTree	78.39%	77.84%	74.33%
SVM	83.8%	84.42%	80.29%

可见如果在特征的选择较为合理的情况（即基于 Gini 的特征选择中）SVM 的效果会随之提升，而 DTree 的提升作用却不太明显。分析他们的实现过程我们可以知道 SVM 的实现中为了能够满足一些分类效果不好的维度的需求会做较大的松弛，这样必然会带来相应效果的下降，而 DTree 则无论特征的多少都会在训练时按需取用，对于一些分类效果不好特征，DTree 会在训练时不再选择，此时特征的降低对于效果的提升作业也就不再明显了。

3）Bagging 对三类学习器的影响分析

<1> Bagging+SVM

我们主要完成了 SVM 的两次训练（分别对应于 20 个 SVM 模型和 50 个 SVM 模型）得到了如下结果：

	RMSE	Rate
单一 SVM 模型	0.33100	/
20 个 SVM 的 Bagging	0.33282	-0.55%
50 个 SVM 的 Bagging	0.33102	基本不变

我们可以看到 Bagging 的方法对于 SVM 的提升是很有限的，深入分析其中的原因可知 Bagging 很强调通过 Bagging 得到的不同分类模型的异质性，如果所有的分类模型都是一样的，那么他们的投票效果自然不会很好。而通过了解 SVM 的训练过程我们可以知道，只有支撑向量对应的 Sample 对于 SVM 而言是有意义的，所以 SVM 对于自己的训练集合的敏感度是不高的，除非是支撑向量被修改了，否则训练结果是不会动的，但是我们通过 Bagging 的方法，绝大多数情况下抽到的点都是非支撑向量，这样训练得到的不同的 SVM 模型是很类似，所以他们的异质性就体现的不是很好，所以训练得到 SVM 模型也和单一的差不多了，甚至有时还会有轻微的下降现象。

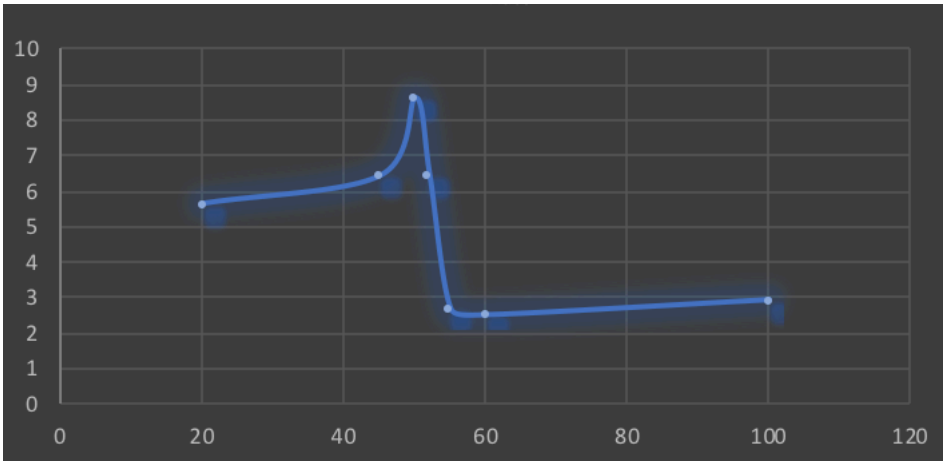
<2>Bagging+DTree

我们使用了 TreeDic（经过 Gini 特征抽取处理以后的字典）构造特征，进行训练得到了如下结果

	RMSE	Rate
单一 DTree 模型	0.27063	/
20 个 DTree 的 Bagging	0.25533	+5.65%
45 个 DTree 的 Bagging	0.2532	+6.44%
50 个 DTree 的 Bagging	0.24726	+8.64%
52 个 DTree 的 Bagging	0.2531	+6.48%
55 个 DTree 的 Bagging	0.26339	+2.68%
60 个 DTree 的 Bagging	0.26384	+2.51%
100 个 DTree 的 Bagging	0.26269	+2.93%

可以看到 DTree+Bagging 的效果是很好的，经过分析我们也发现，和 SVM 不同 DTree 在训练的过程中很重视全局的样本特征，所以我们将其训练集合略作改动（比如通过 Bagging 的方法抽样得到不同的训练集合）DTree 会有很大的变化，所以 DTree 对于训练集合是敏感的，因此它的 Bagging 效果就会很好。

我们将集成的 DTree 的数目关于最终的 Rate 做图如下：



可见在某一个集成度上会出现一个效果的骤降，我认为这个骤降的来源有很多：

1. 可能这本身就是一个随机性的事件，因为 Bagging 的过程本身就有不好的随机性成分

2. 有可能在一些很难分的点上存在一个集成度的阈值，在小于某一个集成度时，由于总的学习器还不太多，总有一些学习器在这些点上做对了，那么他们的决定就会很重要，如果高于某一个集成度，大部分学习器都做失败了，那么这些点的分类就会失败。

经过试验我们选定了最佳集成度为 50

此外还有一个需要讨论的就是在 Bagging 中决策树的特征数量的影响，经过试验，我们发现在大样本集上的决策树进行 Bagging 时的效果往往要比小样本集上的效果更好，如下是使用 50 作为集成度，分别在大样本集上（1w+的原始特征）和小样本集上（1161 基于 Gini 特征方法降维的特征）实验的结果：

	RMSE	Rate
单一 DTree 模型	0.27063	/
50 个小样本 DTree	0.24726	+8.64%
50 个大样本 DTree	0.24106	+10.93%

经过观察发现大样本的 DTree 效果会更好，换句话说，在 Bagging 问题中决策树在他通过 Bagging 抽样获得数据集上拟合的越充分越好。

<3> Bagging+KNN

Bagging 对 KNN 方法有一定的提升，如下表所示：

	RMSE	Rate
单一 KNN 模型	0.54683	/
20 个 KNN 的 Bagging	0.51314	+6.16%

根据我们之前的分析，由于 KNN 对于数据集的变化也比较敏感（即数据集上的变化能够在很大程度上影响到某一些点的 N 近邻，因此 Bagging 对于 KNN 的提升也是很明显的。

4）Boosting 对于两类分类模型的影响

在上一节中我们分析了 Bagging 对于 DTree，SVM，KNN 的影响，这一节中我们来讨论 Boosting 的作用，我们主要讨论 DTree，SVM 两个模型。

<1> Boosting+SVM：

我们分别尝试了不同的采样方法，集成了不同数量的 SVM 分类器的 Boosting 实验，得到了如下结果：

	RMSE	Rate
单一 SVM 模型	0.33100	/
5 个 SVM 的 Boosting(EqualSample)	0.31218	+5.69%
10 个 SVM 的 Boosting(EqualSample)	0.32267	+2.52%
20 个 SVM 的 Boosting(EqualSample)	0.33008	+0.28%
5 个 SVM 的 Boosting(ExtendSample)	0.35312	-6.68%

可以看出 Boosting 和 Bagging 的实验结果有很大的不同，其中最显著的一点在于 Boosting 方法的集成度越高，效果越差，进行简单的分析可知，在 Boosting 的过程中，前一次学对的样本在后一次的权重会降低，而错误的样本的权重则会增加，在我们的 EqualSample 的实现中，后一次学习中抽到的基本上都是错误的样本，同时由于前一次学对的样本肯定要比学错的样本多，这样之后的每一次学习得到的样本集就会变小，所以之后的学习效果就会变差，如果集成度太高，则相对来说效果较差的学习模型的数目就会增加，所以最终的结果就会变差。

另一方面，我们可以看见 ExtendSample 的效果并没有 EqualSample 好，可能是因为 ExtendSample 对应的样本数量太多，这样在 SVM 的实现中需要为一些不可分的样本做的松弛就会太大，所以最后得到的模型效果就会变差。

<2> Boosting+DTree

由于 sklearn 中的决策树自带 Sample_weight 的属性，因此我们不需要通过重采样的方法来模拟权重的实现，但是对于决策树来说在进行 Boosting 时又会出现新的问题。Boosting 训练的逐轮推进是基于上一轮的错误率来的，这也就意味着为了能够给后面的模型更大的可训练空间，前面的模型不能学习的太过于充分，对于决策树中就是要应当适当的限制它的深度或者采用较小的字典集。

下面分别限制 DTree 的深度在 150，100 以及采用较小特征集（TreeDic.txt）的结果（集成度均为 10）

	RMSE	Rate
单一 DTree 模型	0.27063	/
深度限制为 100	0.26269	+2.93%
采用较少的特征	0.27626	-2.08%
深度限制为 150	0.29524	-9.09%

可见限制一定的深度往往能够达到不错的效果，而且相比于限制 150 层，限制 100 层的效果有很大的提升，采用较好的特征也是一种很好的方法，效果介于限制深度为 100 层和 150 层之间。

在 Boosting 中，决策树的集成度同样不能太高，否则效果会有很大幅的下降，但如果集成度太低也不太能够体现出 Boosting 的效果，经过参数调优我们发现 10 是一个不错的选择，相关试验结果如下（限制深度为 100）：

	RMSE	Rate
单一 DTree 模型	0.27063	/
5 个 DTree 的 Boosting	0.27507	-1.64%
10 个 DTree 的 Boosting	0.26269	+2.93%
20 个 DTree 的 Boosting	0.30827	-13.9%

5. 收获与总结

本次实验让我学会了 SVM 和 DTree 的基本使用方法，同时还理解了 Bagging 和 Boosting 的主要思路，同时在参数调优的过程中也有很多的感慨和收获。

我在上周六时就完成了本实验的大部分内容，有了大概 3，4 天整理数据同时在此基础上完成了这份非常详尽的分析报告。在这个过程中也学会了不少学术论文的写作技巧。