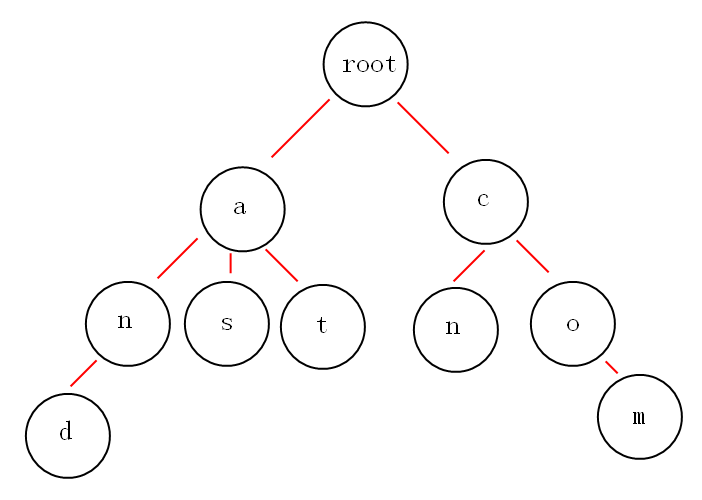
# 图算法

## Trie树

字典树，前缀树

应用：词频统计、前缀匹配



1：定义trie树节点

     为了方便，我也采用纯英文字母，我们知道字母有26个，那么我们构建的trie树就是一个26叉树，每个节点包含26个子节点。

1 #region Trie树节点

2 /// <summary>

3 /// Trie树节点

4 /// </summary>

5 public class TrieNode

6 {

7 /// <summary>

8 /// 26个字符，也就是26叉树

9 /// </summary>

10 public TrieNode[] childNodes;

11

12 /// <summary>

13 /// 词频统计

14 /// </summary>

15 public int freq;

16

17 /// <summary>

18 /// 记录该节点的字符

19 /// </summary>

20 public char nodeChar;

21

22 /// <summary>

23 /// 插入记录时的编码id

24 /// </summary>

25 public HashSet<int> hashSet = new HashSet<int>();

26

27 /// <summary>

28 /// 初始化

29 /// </summary>

30 public TrieNode()

31 {

32 childNodes = new TrieNode[26];

33 freq = 0;

34 }

35 }

36 #endregion

2: 添加操作

     既然是26叉树，那么当前节点的后续子节点是放在当前节点的哪一叉中，也就是放在childNodes中哪一个位置，这里我们采用

      int k = word[0] - 'a'来计算位置。

1 /// <summary>

2 /// 插入操作

3 /// </summary>

4 /// <param name="root"></param>

5 /// <param name="s"></param>

6 public void AddTrieNode(ref TrieNode root, string word, int id)

7 {

8 if (word.Length == 0)

9 return;

10

11 //求字符地址，方便将该字符放入到26叉树中的哪一叉中

12 int k = word[0] - 'a';

13

14 //如果该叉树为空，则初始化

15 if (root.childNodes[k] == null)

16 {

17 root.childNodes[k] = new TrieNode();

18

19 //记录下字符

20 root.childNodes[k].nodeChar = word[0];

21 }

22

23 //该id途径的节点

24 root.childNodes[k].hashSet.Add(id);

25

26 var nextWord = word.Substring(1);

27

28 //说明是最后一个字符，统计该词出现的次数

29 if (nextWord.Length == 0)

30 root.childNodes[k].freq++;

31

32 AddTrieNode(ref root.childNodes[k], nextWord, id);

33 }

34 #endregion

3：删除操作

     删除操作中，我们不仅要删除该节点的字符串编号，还要对词频减一操作。

/// <summary>

/// 删除操作

/// </summary>

/// <param name="root"></param>

/// <param name="newWord"></param>

/// <param name="oldWord"></param>

/// <param name="id"></param>

public void DeleteTrieNode(ref TrieNode root, string word, int id)

{

if (word.Length == 0)

return;

//求字符地址，方便将该字符放入到26叉树种的哪一颗树中

int k = word[0] - 'a';

//如果该叉树为空,则说明没有找到要删除的点

if (root.childNodes[k] == null)

return;

var nextWord = word.Substring(1);

//如果是最后一个单词，则减去词频

if (word.Length == 0 && root.childNodes[k].freq > 0)

root.childNodes[k].freq--;

//删除途经节点

root.childNodes[k].hashSet.Remove(id);

DeleteTrieNode(ref root.childNodes[k], nextWord, id);

}

## SlopeOne

协同推荐

应用：同类推荐

SlopeOne的思想很简单，就是用均值化的思想来掩盖个体的打分差异，举个例子说明一下：

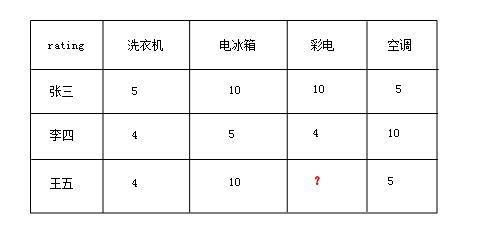


在这个图中，系统该如何计算“王五“对”电冰箱“的打分值呢？刚才我们也说了，slopeone是采用均值化的思想,也就是：R王五 =4-{[(5-10)+(4-5)]/2}=7 。

下面我们看看多于两项的商品，如何计算打分值。

rb = (n \* (ra - R(A->B)) + m \* (rc - R(C->B)))/(m+n)

注意： a,b,c 代表“商品”。ra 代表“商品的打分值”。ra->b  代表“A组到B组的平均差（均值化）”。m,n 代表人数。



根据公式，我们来算一下。

r王五 = (2 \* (4 - R(洗衣机->彩电)) + 2 \* (10 - R(电冰箱->彩电))+ 2 \* (5 - R(空调->彩电)))/(2+2+2)=6.8

是的，slopeOne就是这么简单，实战效果非常不错。

二：实现

1：定义一个评分类Rating。

1 /// <summary>

2 /// 评分实体类

3 /// </summary>

4 public class Rating

5 {

6 /// <summary>

7 /// 记录差值

8 /// </summary>

9 public float Value { get; set; }

10

11 /// <summary>

12 /// 记录评分人数，方便公式中的 m 和 n 的值

13 /// </summary>

14 public int Freq { get; set; }

15

16 /// <summary>

17 /// 记录打分用户的ID

18 /// </summary>

19 public HashSet<int> hash\_user = new HashSet<int>();

20

21 /// <summary>

22 /// 平均值

23 /// </summary>

24 public float AverageValue

25 {

26 get { return Value / Freq; }

27 }

28 }

2： 定义一个产品类

1 /// <summary>

2 /// 产品类

3 /// </summary>

4 public class Product

5 {

6 public int ProductID { get; set; }

7

8 public string ProductName { get; set; }

9

10 /// <summary>

11 /// 对产品的打分

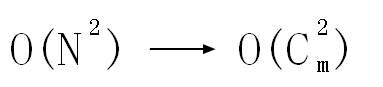
12 /// </summary>

13 public float Score { get; set; }

14 }

3：SlopeOne类

     参考了网络上的例子，将二维矩阵做成线性表，有效的降低了空间复杂度。



1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5

6 namespace SupportCenter.Test

7 {

8 #region Slope One 算法

9 /// <summary>

10 /// Slope One 算法

11 /// </summary>

12 public class SlopeOne

13 {

14 /// <summary>

15 /// 评分系统

16 /// </summary>

17 public static Dictionary<int, Product> dicRatingSystem = new Dictionary<int, Product>();

18

19 public Dictionary<string, Rating> dic\_Martix = new Dictionary<string, Rating>();

20

21 public HashSet<int> hash\_items = new HashSet<int>();

22

23 #region 接收一个用户的打分记录

24 /// <summary>

25 /// 接收一个用户的打分记录

26 /// </summary>

27 /// <param name="userRatings"></param>

28 public void AddUserRatings(IDictionary<int, List<Product>> userRatings)

29 {

30 foreach (var user1 in userRatings)

31 {

32 //遍历所有的Item

33 foreach (var item1 in user1.Value)

34 {

35 //该产品的编号（具有唯一性）

36 int item1Id = item1.ProductID;

37

38 //该项目的评分

39 float item1Rating = item1.Score;

40

41 //将产品编号字存放在hash表中

42 hash\_items.Add(item1.ProductID);

43

44 foreach (var user2 in userRatings)

45 {

46 //再次遍历item，用于计算俩俩 Item 之间的差值

47 foreach (var item2 in user2.Value)

48 {

49 //过滤掉同名的项目

50 if (item2.ProductID <= item1Id)

51 continue;

52

53 //该产品的名字

54 int item2Id = item2.ProductID;

55

56 //该项目的评分

57 float item2Rating = item2.Score;

58

59 Rating ratingDiff;

60

61 //用表的形式构建矩阵

62 var key = Tools.GetKey(item1Id, item2Id);

63

64 //将俩俩 Item 的差值 存放到 Rating 中

65 if (dic\_Martix.Keys.Contains(key))

66 ratingDiff = dic\_Martix[key];

67 else

68 {

69 ratingDiff = new Rating();

70 dic\_Martix[key] = ratingDiff;

71 }

72

73 //方便以后以后userrating的编辑操作，（add)

74 if (!ratingDiff.hash\_user.Contains(user1.Key))

75 {

76 //value保存差值

77 ratingDiff.Value += item1Rating - item2Rating;

78

79 //说明计算过一次

80 ratingDiff.Freq += 1;

81 }

82

83 //记录操作人的ID，方便以后再次添加评分

84 ratingDiff.hash\_user.Add(user1.Key);

85 }

86 }

87 }

88 }

89 }

90 #endregion

91

92 #region 根据矩阵的值，预测出该Rating中的值

93 /// <summary>

94 /// 根据矩阵的值，预测出该Rating中的值

95 /// </summary>

96 /// <param name="userRatings"></param>

97 /// <returns></returns>

98 public IDictionary<int, float> Predict(List<Product> userRatings)

99 {

100 Dictionary<int, float> predictions = new Dictionary<int, float>();

101

102 var productIDs = userRatings.Select(i => i.ProductID).ToList();

103

104 //循环遍历\_Items中所有的Items

105 foreach (var itemId in this.hash\_items)

106 {

107 //过滤掉不需要计算的产品编号

108 if (productIDs.Contains(itemId))

109 continue;

110

111 Rating itemRating = new Rating();

112

113 // 内层遍历userRatings

114 foreach (var userRating in userRatings)

115 {

116 if (userRating.ProductID == itemId)

117 continue;

118

119 int inputItemId = userRating.ProductID;

120

121 //获取该key对应项目的两组AVG的值

122 var key = Tools.GetKey(itemId, inputItemId);

123

124 if (dic\_Martix.Keys.Contains(key))

125 {

126 Rating diff = dic\_Martix[key];

127

128 //关键点：运用公式求解（这边为了节省空间，对角线两侧的值呈现奇函数的特性）

129 itemRating.Value += diff.Freq \* (userRating.Score + diff.AverageValue \* ((itemId < inputItemId) ? 1 : -1));

130

131 //关键点：运用公式求解 累计每两组的人数

132 itemRating.Freq += diff.Freq;

133 }

134 }

135

136 predictions.Add(itemId, itemRating.AverageValue);

137 }

138

139 return predictions;

140 }

141 #endregion

142 }

143 #endregion

144

145 #region 工具类

146 /// <summary>

147 /// 工具类

148 /// </summary>

149 public class Tools

150 {

151 public static string GetKey(int Item1Id, int Item2Id)

152 {

153 return (Item1Id < Item2Id) ? Item1Id + "->" + Item2Id : Item2Id + "->" + Item1Id;

154 }

155 }

156 #endregion

157 }