|  |
| --- |
| 1. package com.electionpredictor;  2.  3. import java.io.IOException;  4. import java.util.ArrayList;  5. import java.util.List;  6.  7. import org.apache.commons.cli.BasicParser;  8. import org.apache.commons.cli.CommandLine;  9. import org.apache.commons.cli.CommandLineParser;  10. import org.apache.commons.cli.Option;  11. import org.apache.commons.cli.OptionBuilder;  12. import org.apache.commons.cli.Options;  13. import org.apache.commons.cli.ParseException;  14.  15. import com.electionpredictor.file.FileHandler;  16. import com.electionpredictor.instance.Election;  17. import com.electionpredictor.instance.Instance;  18. import com.electionpredictor.instance.PartyID;  19. import com.electionpredictor.predictors.ElectionRange;  20. import com.electionpredictor.predictors.PartyRange;  21. import com.electionpredictor.predictors.Prediction;  22. import com.electionpredictor.ui.Application;  23.  24. /\*\*  25. \* Get the ball rolling!  26. \*  27. \* @author Niels Stchedroff  28. \*/  29. public class KickOff  30. {  31. private static final String AP = "ap";  32. private static final String F = "f";  33. private static final String L = "l";  34. private static Options mOptions;  35. private static final String P1 = "p1";  36. private static final String P2 = "p2";  37. private static final int PARTY\_RANGE\_ARGUMENT\_COUNT = 4;  38.  39. /\*\*  40. \* @param args  41. \* @throws IOException  42. \*/  43. public static void main(final String[] args) throws IOException  44. {  45. if ((args == null) || (args.length == 0))  46. {  47. Application.runApplication();  48. }  49. else  50. {  51. KickOff.runFromCommandLine(args);  52. }  53. }  54.  55. private static PartyRange buildRangeFromOption(final Instance inst, final Option option)  56. {  57. if (inst == null) { throw new IllegalArgumentException("Instance is null!"); }  58.  59. if (option == null) { throw new IllegalArgumentException("Option is null!"); }  60.  61. final String[] values = option.getValues();  62. PartyRange result;  63. PartyID partyName = null;  64. double min = 0.0;  65. double max = 0.0;  66. int intervals = 0;  67.  68. if ((values == null) || (values.length != KickOff.PARTY\_RANGE\_ARGUMENT\_COUNT)) { throw new IllegalArgumentException("Option "  69. + option.getOpt() + " must have " + option.getArgs() + " arguments"); }  70.  71. try  72. {  73. partyName = PartyID.valueOf(option.getValues()[0]);  74. }  75. catch (final IllegalArgumentException exp)  76. {  77. throw new IllegalArgumentException("Option " + option.getOpt() + " must have a valid party - value was " + option.getValues()[0], exp);  78. }  79.  80. try  81. {  82. min = Double.valueOf(option.getValues()[1]);  83. }  84. catch (final NumberFormatException numExp)  85. {  86. throw new NumberFormatException("Party " + partyName + " must have a valid number for minimum support! '" + option.getValues()[1]  87. + "' is not valid.");  88. }  89.  90. try  91. {  92. max = Double.valueOf(option.getValues()[2]);  93. }  94. catch (final NumberFormatException numExp2)  95. {  96. throw new NumberFormatException("Party " + partyName + " must have a valid number for maximum support! '" + option.getValues()[2]  97. + "' is not valid.");  98. }  99.  100. try  101. {  102. intervals = Integer.valueOf(option.getValues()[3]);  103. }  104. catch (final NumberFormatException numExp2)  105. {  106. throw new NumberFormatException("Party " + partyName + " must have a valid number for intervals! '" + option.getValues()[3]  107. + "' is not valid.");  108. }  109.  110. if (min < 0) { throw new IllegalArgumentException("Party " + partyName + " must have a positive minimum support!"); }  111.  112. if (max < min) { throw new IllegalArgumentException("Party " + partyName + " must have a max that is greater than min!"); }  113.  114. if ((min > 100) || (max > 100)) { throw new IllegalArgumentException("Party " + partyName + " must have max and imn that are less than 100%!"); }  115.  116. if (intervals < 0) { throw new IllegalArgumentException("Party " + partyName + " must have a number of intervals that is non-zero!"); }  117.  118. result = new PartyRange(inst.getParty(partyName), min / 100, max / 100, intervals);  119.  120. return result;  121. }  122.  123. private static PartyID getPartyFromOption(final Option option)  124. {  125. if (option == null) { throw new IllegalArgumentException("Option is null!"); }  126.  127. final String[] values = option.getValues();  128. PartyID partyName = null;  129.  130. if ((values == null) || (values.length != 1)) { throw new IllegalArgumentException("Option " + option.getOpt() + " must have "  131. + option.getArgs() + " arguments"); }  132.  133. try  134. {  135. partyName = PartyID.valueOf(option.getValues()[0]);  136. }  137. catch (final IllegalArgumentException exp)  138. {  139. throw new IllegalArgumentException("Option " + option.getOpt() + " must have a valid party - value was " + option.getValues()[0], exp);  140. }  141.  142. return partyName;  143. }  144.  145. private static void runFromCommandLine(final String[] args) throws IOException  146. {  147. KickOff.setupArguments();  148. final Instance instance = new Instance(new Application());  149. final CommandLineParser parser = new BasicParser();  150. PartyRange partyA = null;  151. PartyRange partyB = null;  152. PartyID adjustmentParty = null;  153. String filePath = null;  154. String dataDirectory = null;  155.  156. try  157. {  158. final CommandLine cmd = parser.parse(KickOff.mOptions, args);  159. final Option[] options = cmd.getOptions();  160.  161. for (final Option option : options)  162. {  163. if (option.getOpt().equalsIgnoreCase(KickOff.P1))  164. {  165. partyA = KickOff.buildRangeFromOption(instance, option);  166. }  167.  168. if (option.getOpt().equalsIgnoreCase(KickOff.P2))  169. {  170. partyB = KickOff.buildRangeFromOption(instance, option);  171. }  172.  173. if (option.getOpt().equalsIgnoreCase(KickOff.AP))  174. {  175. adjustmentParty = KickOff.getPartyFromOption(option);  176. }  177.  178. if (option.getOpt().equalsIgnoreCase(KickOff.F))  179. {  180. filePath = option.getValue();  181. }  182.  183. if (option.getOpt().equalsIgnoreCase(KickOff.L))  184. {  185. dataDirectory = option.getValue();  186. }  187. }  188.  189. final Election election = (Election) instance.readElection(dataDirectory, dataDirectory);  190. final List<PartyRange> parties = new ArrayList<PartyRange>();  191. parties.add(partyA);  192. parties.add(partyB);  193.  194. final ElectionRange er = new ElectionRange(instance, election, parties, instance.getParty(adjustmentParty));  195.  196. final ArrayList<String[]> results = instance.buildAddativeModelRangeData(null, er, null);  197.  198. results.add(0, Prediction.displayPredictionHeader(instance.getPartyStore().getPartyList()));  199.  200. FileHandler.writeDataToFile(filePath, results);  201. }  202. catch (final ParseException e1)  203. {  204. // TODO Auto-generated catch block  205. e1.printStackTrace();  206. }  207. }  208.  209. /\*\*  210. \* Construct the  211. \* Construct the arguments for the command line  212. \*/  213. private static void setupArguments()  214. {  215. // create Options object  216. KickOff.mOptions = new Options();  217. KickOff.mOptions.addOption(OptionBuilder.hasArgs(KickOff.PARTY\_RANGE\_ARGUMENT\_COUNT).isRequired().withValueSeparator(',').withDescription(  218. "party 1 name").create(KickOff.P1));  219. KickOff.mOptions.addOption(OptionBuilder.hasArgs(KickOff.PARTY\_RANGE\_ARGUMENT\_COUNT).isRequired().withValueSeparator(',').withDescription(  220. "party 2 name").create(KickOff.P2));  221. KickOff.mOptions.addOption(OptionBuilder.hasArgs(1).isRequired().withValueSeparator(',').withDescription("adjustment party name").create(  222. KickOff.AP));  223. KickOff.mOptions.addOption(OptionBuilder.hasArgs(1).isRequired().withValueSeparator(',').withDescription("output file name")  224. .create(KickOff.F));  225. KickOff.mOptions.addOption(OptionBuilder.hasArgs(1).isRequired().withValueSeparator(',').withDescription("last election").create(KickOff.L));  226. }  227. } |