

git_comments:

1. Ignore, and break.
2. Ignore, and break.

git_commits:

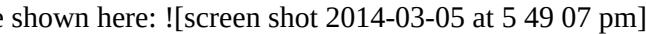
1. **summary:** [SPARK-32034][SQL] Port HIVE-14817: Shutdown the SessionManager timeoutChecker thread properly upon shutdown
message: [SPARK-32034][SQL] Port HIVE-14817: Shutdown the SessionManager timeoutChecker thread properly upon shutdown #### What changes were proposed in this pull request? This PR port <https://issues.apache.org/jira/browse/HIVE-14817> for spark thrift server. #### Why are the changes needed? When stopping the HiveServer2, the non-daemon thread stops the server from terminating ``sql "HiveServer2-Background-Pool: Thread-79" #79 prio=5 os_prio=31 tid=0x00007fde26138800 nid=0x13713 waiting on condition [0x0000700010c32000] java.lang.Thread.State: TIMED_WAITING (sleeping) at java.lang.Thread.sleep(Native Method) at org.apache.hive.service.cli.session.SessionManager\$1.sleepInterval(SessionManager.java:178) at org.apache.hive.service.cli.session.SessionManager\$1.run(SessionManager.java:156) at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149) at java.util.concurrent.ThreadPoolExecutor\$Worker.run(ThreadPoolExecutor.java:624) at java.lang.Thread.run(Thread.java:748) ``` Here is an example to reproduce: <https://github.com/yaoqinn/kyuubi/blob/master/kyuubi-spark-sql-engine/src/main/scala/org/apache/kyuubi/spark/SparkSQLEngineApp.scala> Also, it causes issues as HIVE-14817 described which #### Does this PR introduce _any_ user-facing change? NO #### How was this patch tested? Passing Jenkins Closes #28870 from yaoqinn/SPARK-32034. Authored-by: Kent Yao <yaoqinn@hotmail.com> Signed-off-by: Dongjoon Hyun <dongjoon@apache.org> (cherry picked from commit 9f8e15bb2e2189812ee34e3e64baede0d799ba76) Signed-off-by: Dongjoon Hyun <dongjoon@apache.org>

github_issues:**github_issues_comments:****github_pulls:**

1. **title:** MLI-1 Decision Trees
body: Joint work with @hirakendu, @etrain, @atalwalkar and @harsha2010. Key features: - Supports binary classification and regression - Supports gini, entropy and variance for information gain calculation - Supports both continuous and categorical features The algorithm has gone through several development iterations over the last few months leading to a highly optimized implementation. Optimizations include: 1. Level-wise training to reduce passes over the entire dataset. 2. Bin-wise split calculation to reduce computation overhead. 3. Aggregation over partitions before combining to reduce communication overhead.

github_pulls_comments:

1. Can one of the admins verify this patch?
2. I'm looking forward to this. I have one question just based on the description and not reading the code. Why only binary classification? RDF is inherently amenable to multi-class; you're just storing a distribution over N classes and computing entropy over N classes rather than 2. Also does this support evaluating feature importance? even the simplistic way done by the likes of scikit-learn where you just evaluate which features touch the most examples as the pass through trees. Those are to me two key features for RDF. (And actual classification of new examples, but I take that for granted)
3. Thanks Sean. Multi-class classification and feature importances are important features that will be added soon. We implemented a minimal feature set since we wanted to focus on functional accuracy and (weak and strong) scaling. Now that we are satisfied on that front, I am sure these features will be added soon. It's a fairly big PR in terms of code size so I prefer to avoid adding any more features to the basic implementation. Also, we have plans to add ensemble trees (random decision forests, boosting, etc.) soon to mllib. Finally, even though mllib lacks this functionality just yet, one could always implement a bank of one-versus-all classifiers as a workaround to handle the multi-class classification problem. At the same time, I agree its important to add this functionality to the classification algorithm itself and will be added soon.
4. This PR is the result of several iterations on the idea of wanting to build fast decision trees for Spark. To offer a little more color on the design here, the key ideas are: 1. Histograms to compute best split statistics. We quantize input attributes and compute best splits via relatively cheap operations on histograms. This is a precision/speed

tradeoff in that we give up the ability to find "perfect" split points (by the C4.5 algorithm), but still find very close ones. 2. Minimize total passes over the data by training models "level-wise". At each level of the tree, we evaluate which node a point belongs in, and allow the point to contribute to the statistics (histogram bins) for that node only. While the number of histograms is thus exponential with tree depth, these data structures are relatively small and we expect that very deep trees are relatively uncommon (especially in the ensemble case). This also means that we need only cache the working set once, instead of partitioning the data and caching the partitions. 3. Want to support arbitrary split statistics (e.g. Gini, Entropy, Variance), and both categorical and continuous input and response values - I believe that the current design allows for this efficiently. I tested the code in this PR for scalability on clusters with 2 slaves (m2.xlarge nodes) to a cluster with 16 slaves of the same variety. Master was also an m2.xlarge. Data that was thrown at this PR was generated from the LinearDataGenerator class. Ranging in size from 10m points to 50m points, and dimensions d=10 to d=50 - this equates to 700mb of training data up to 18gb of training data, all models were trained to a maximum depth of 10. Total runtime on 16 machines ranged from under 5 minutes for the 10m_10 experiment to 36 minutes for the 50m_50 experiment. Scalability results are shown here: 

(<https://f.cloud.github.com/assets/1326181/2341114/92d71ce6-a4d1-11e3-8efd-82d57dc68c27.png>) On the x-axis we have number of machines (2..16) and the y-axis is speedup. Each box in the grid represents a different dataset. Ideal scaling would be 8x. There are several things worth pointing out on this chart: 1. There are some cases where the 2-machine experiments didn't finish (the last 2 30m experiments and all of the 40m and 50m experiments), and in those cases ideal speedup is 4x. 2. There are also a couple of cases where we appear to be doing better than ideal, but those are cases where GC overhead was substantial in the 2 machine case, so I don't consider those apples-to-apples. 3. Overall, through this strategy of parallelization, we see an average of a 6.6x speedup for 16 nodes vs. 2 (17% deterioration from ideal) with a standard deviation in speedup of 0.7 across these experiments for those observations that I'm calling "fair".

5. @manishamde Do you mind updating the code style first to make it easy for people who want to review the code? I will mark a few examples. We also need a Spark JIRA ticket for this PR. @etrain It would be great if the optimization you mentioned is documented somewhere in the code or in the guide. It helps understand the code and the comments here will be ignored after this PR gets merged.
6. Jenkins, add to whitelist and test this please
7. Merged build triggered.
8. Merged build started.
9. Merged build finished.
10. One or more automated tests failed Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13028/>
11. That is Jenkins complaining about the style BTW, hopefully should be easy to fix. You can run sbt/sbt scalastyle to run the same tests locally.
12. @mengxr @hsaputra Thanks for the code style comments. I have made a lot of effort to document the code. I guess I still need to make the code consistent with the Spark style guidelines. I will also note down the optimizations in the code for future reference.
13. @manishamde thank you for decision tree contribution and the detail comments/ documentation in the PR :) Looking forward to review and seeing this as part of Spark MLlib.
14. Merged build triggered.
15. Merged build started.
16. Merged build finished.
17. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13041/>
18. Merged build triggered.
19. Merged build started.
20. Merged build finished.
21. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13043/>
22. Merged build triggered.
23. Merged build started.
24. Merged build finished.
25. One or more automated tests failed Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13091/>
26. Merged build triggered.
27. Merged build started.
28. Merged build finished.
29. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13094/>
30. Thanks for another round of reviews @mengxr. I will fix these style issues, answer your comments and add more documentation around the optimizations later today. In general, I wonder whether we could add these checks to scalastyle to be cognizant of the style rules during development. May be we could also update the scala code style

guide to reflect these requirements. For somebody, who likes consistency but codes in multiple languages/codebases in a day, it's good to have these forced guidelines for easier development. Also, it reduces the style-checking during the review which would reduce the burden on the reviewers as well. Thanks again for the detailed reviews.

31. @manishamde Thanks for updating the code style and adding more docs! I made a first pass over the code. For the code style, we do not have a good style checker for Scala. @rxin can tell more about style checking. However, it is easy to learn Spark's code style through the code review and make your code style consistent in the next update. Please see my comments for some examples and update similar code in other places. For the implementation, I have the following suggestions: 1. Regression or Classification is checked in many places. It would be nice to create a DecisionTree base class and make RegressionTree and ClassificationTree two subclasses of it. 2. For loops are used in some performance critical code. This should be replaced by "while", which is much faster than "for" in Scala. 3. Several nested methods are used in findBestSplits. It feels safe to see some unit tests for them. 4. The threshold for classification is set at 0.5. This should be configurable. I will try to make a second pass focusing on the algorithm later today. In the meanwhile, would you please update the remaining code style problems and the for loops? Thanks!
32. @mengxr Thanks for such a detailed review. The code is already in a much better shape after incorporating your suggested changes. I have fixed the straightforward code style issues and will commit shortly. I still need to make more code style changes and document the findBestSplit method in detail. I need some more time to think about your implementation suggestions. I also need to seriously consider @srowen's suggestion about adding multi-class classification in this version of the PR. I will get back to you on this tomorrow.
33. Merged build triggered.
34. Merged build started.
35. Merged build triggered.
36. Merged build started.
37. Merged build finished.
38. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13105/>
39. Merged build finished.
40. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13106/>
41. Merged build triggered.
42. Merged build started.
43. Merged build finished.
44. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13148/>
45. Merged build triggered.
46. Merged build started.
47. Merged build finished.
48. One or more automated tests failed Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13149/>
49. Jenkins, retest this please.
50. Merged build triggered.
51. Merged build started.
52. Merged build finished.
53. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13172/>
54. Merged build triggered.
55. Merged build started.
56. Merged build finished.
57. One or more automated tests failed Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13207/>
58. Jenkins, retest this please.
59. Merged build triggered.
60. Merged build started.
61. @manishamde Please let me know if this is read for another pass. Thanks!
62. @mengxr Almost there! Needs some more documentation. I hope to finish soon.
63. Merged build finished.
64. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13210/>
65. @mengxr It's ready for another pass. :-)
66. Merged build triggered.
67. Merged build started.
68. Merged build finished.

69. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13350/>
70. @manishamde I sent you a PR containing mostly code style updates at
<https://github.com/manishamde/spark/pull/1> Please take a look and merge it to this PR if it looks good to you.
71. Merged build triggered.
72. Merged build started.
73. @mengxr Thanks a lot! LGTM. Merged.
74. Merged build triggered.
75. Merged build started.
76. Merged build finished.
77. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13358/>
78. Merged build finished.
79. One or more automated tests failed Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13361/>
80. Merged build triggered.
81. Merged build started.
82. Merged build finished.
83. One or more automated tests failed Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13363/>
84. Merged build triggered.
85. Merged build started.
86. Merged build finished.
87. One or more automated tests failed Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13378/>
88. Merged build triggered.
89. Merged build started.
90. Merged build finished.
91. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13419/>
92. Can one of the admins verify this patch?
93. Jenkins, this is okay to test. Jenkins, test this please.
94. Merged build triggered. One or more automated tests failed.
95. Merged build started. One or more automated tests failed.
96. Merged build finished. One or more automated tests failed.
97. One or more automated tests failed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/Spark-prb/2/>
98. Merged build triggered. One or more automated tests failed.
99. Merged build started. One or more automated tests failed.
100. Merged build finished. Some tests failed or tests have not started running yet.
101. Some tests failed or tests have not started running yet. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13536/>
102. Jenkins, retest this please.
103. Merged build triggered. Build is starting -or- tests failed to complete.
104. Merged build started. Build is starting -or- tests failed to complete.
105. @manishamde I sent some minor code style updates to your repo. Please take a look and merge it into this PR.
Thanks!
106. Merged build finished. All automated tests passed.
107. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13609/>
108. @mengxr I am not sure the merge of your PR on my repo went through. I might have closed the PR accidentally and I can't find a way to undo it. Could you please re-send your latest PR?
109. @manishamde Please merge the changes at <https://github.com/manishamde/spark/pull/4>
110. Merged build triggered. Build is starting -or- tests failed to complete.
111. Merged build started. Build is starting -or- tests failed to complete.
112. Merged build finished. All automated tests passed.
113. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13614/>
114. Merged build triggered.
115. Merged build started.
116. Merged build finished. All automated tests passed.
117. All automated tests passed. Refer to this link for build results:
<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/13623/>

118. @mateiz This looks good to me and I will mark some APIs developer/experimental after it gets merged. Please help merge it. Thanks @manishamde , @hirakendu , @etrain , @atalwalkar , and @harsha2010 for the work!
119. Excellent, thanks a lot Manish, Ram and others! I've merged this in.
120. Thanks Manish, Evan, Ameet, Ram, and thanks Xiangrui, Matei. It's great to have this PR merged, but I think there is room for a lot of improvement, and I hope this thread is still open for comments. I have some detailed notes from my side, we can track them in a separate place if required. Some design notes to start with: The current design of classes and relationships is good, but I think it would be great if it can be modified slightly to make it similar to the design of other existing algorithms in MLLib and extend the existing interfaces. For example, we can have a `DecisionTreeModel` or that extends the existing `RegressionModel` interface, similar to the existing `RidgeRegressionModel` and `LinearRegressionModel`. Alternatively, we can have separate `ClassificationTreeModel` and `RegressionTreeModel` that extend the existing interfaces `ClassificationModel` and `RegressionModel` respectively. Note that it is important to keep the `Model` as a class, so that we can later compose ensemble and other composite models consisting of multiple `Model` instances. Currently the decision tree `Node` is essentially the `Model`, although I would prefer a wrapper around it and explicitly called `DecisionTreeModel` that implements methods like `predict`, `save`, `explain`, `load` like I remember coming across in MLI. In the file `DecisionTree.scala`, the actual class can be renamed to `DecisionTreeAlgorithm` that extends `Algorithm` interface that implements `train(samples)` and outputs a `RegressionModel`. Note that there is no `Algorithm` interface currently in MLLib, although there may be one in MLI and the closest is `abstract class GeneralizedLinearAlgorithm[M <: GeneralizedLinearModel]`. Modeling `Strategy` should be renamed to `Parameters`. I would prefer a separation between model parameters like depth, minimum gain and algorithm parameters like level-by-level training or caching. The line is blurred for some aspects like quantile strategy, but I am inclined to put those into modeling parameters, so it can be referred to later on. Rule of thumb being that different algorithm parameters should lead to same model (up to randomization effects) for the same model parameters. `Impurity` should be renamed to `Error` or something more technical. Also see my later comments on the need for a generic `Error` Interface that allow easy adaptation of algorithms for specific loss functions. In general, MLLib and MLI should define proper interfaces for `Model`, `Algorithm`, `Parameters` and importantly `Loss` entities, at least for supervised machine learning algorithms and all implementations should adhere to it. Surprisingly, MLLib or MLI currently doesn't have a `Loss` or `Error` interface, the closest being the `Optimizer` interface. There is also a need for portable model output formats, e.g., JSON, that can be used by other programs, possibly written in other languages outside Scala and Java. Can also use an existing format like PMML (not sure if it's widely used). Lastly, there is a need to support standard data formats with optional delimiter parameters - I am sure this a general need for Spark. I understand there has been significant effort before for standardization, would be good to know about the current status.
121. In terms of running time performance, here are some scalability results for a large scale dataset. Results look satisfactory :). The code was tested on a dataset for a binary classification problem. A regression tree with `Variance` (square loss) was trained because it's computationally more intensive. The dataset consists of about `500,000,000` training instances. There are 20 features, all numeric. Although there are categorical features in the dataset and the algorithm implementation supports it, they were not used since the main program doesn't have options. The dataset is of size about 90 GB in plain text format. It consists of 100 part files, each about 900 MB. To _optimize_ number of tasks to align with number of workers, the task split size was chosen to be 160 MB to have 300 tasks. The model training experiments were done on a Yahoo internal Hadoop cluster. The CPUs are of type Intel Xeon 2.4 GHz. The Spark YARN adaptation was used to run on Hadoop cluster. Both master-memory and worker-memory were set to `7500m` and the cluster was under light to moderate load at the time of experiments. The fraction of memory used for caching was `0.3`, leading to about `2 GB` of memory per worker for caching. The number of workers used for various experiments were `20, 30, 60, 100, 150, 300, 600` to evenly align with 600 tasks. Note that with 20 and 30 workers, only `48%` and `78%` of the 90 GB data could be cached in memory. The decision tree of depth 10 was trained and minor code changes were made to record the individual level training times. The training command (with additional JVM settings) used is `` time \ SPARK_JAVA_OPTS="\$\${SPARK_JAVA_OPTS} \ -Dspark.hadoop.mapred.min.split.size=167772160 \ -Dspark.hadoop.mapred.max.split.size=167772160" \ -Dspark.storage.memoryFraction=0.3 \ spark-class org.apache.spark.deploy.yarn.Client \ --queue \${QUEUE} \ --num-workers \${NUM_WORKERS} \ --worker-memory 7500m \ --worker-cores 1 \ --master-memory 7500m \ --jar \${JARS}/spark_mllib_tree.jar \ --class org.apache.spark.mllib.tree.DecisionTree \ --args yarn-standalone \ --args --algo --args Regression \ --args --trainDataDir --args \${DIST_WORK}/train_data_lp.txt \ --args --testDataDir --args \${DIST_WORK}/test_data_0p1pc_lp.txt \ --args --maxDepth --args 10 \ --args --impurity --args Variance \ --args --maxBins --args 100 `` The training times for training each depth and for each choice of number of workers is in the attachments [workers_times.txt] (https://raw.githubusercontent.com/hirakendu/temp/master/spark_mllib_tree_pr79_review/workers_times.txt). The attached graphs demonstrate the scalability in terms of cumulative training times for various depths and various number of workers.)) For obtaining the speed-ups, the training times are compared to those for 60 workers, since the data could not be

cached completely for 20 and 30 workers. For all experiments, the resources requested were fully allocated by the cluster and all experiments ran to completion in their first and only run. As we can see, the scaling is nearly linear for higher depths 9 and 10, across the range of 60 workers to 600 workers, although the slope is less than 1 as expected. For such loads, 60 to 100 workers are a reasonable computing resource and the total training times are about 160 minutes and 100 minutes respectively. Overall, the performance is satisfactory, in particular when trees of depth 4 or 5 are trained for boosting models. But clearly, there is room for improvement in the following sense. The dataset when fully cached takes about 10s for a count operation, whereas the training time for first level that involves simple histogram calculation of three error statistics takes roughly 30 seconds. The error performance in terms of RMSE was verified to be close to that of alternate implementations.

122. Since it's a long review, I have put things together in one place [spark_mllib_tree_pr79_review.md] (https://github.com/hirakendu/temp/blob/master/spark_mllib_tree_pr79_review/spark_mllib_tree_pr79_review.md) so that it's easy to track later.
123. Hi Hirakendu - thanks for all the detailed suggestions and information. I will reply to that separately. One question - you say there are 500,000 examples and this equates to 90GB of raw data. If that's the case, this works out to ~200KB per example - is that right or are you off by an order of magnitude in either the number of features or the number of data points? Or are we throwing a bunch of data out before fitting?
124. Thanks for noticing the typo, it's 500 million examples. Corrected it. Coincidentally, the in-memory storage size is also around 90 GB when cached.
125. @hirakendu Thanks a lot for the detailed comments and feedback. Yes, we have a responsibility to keep improving the trees going forward so getting additional feedback is awesome. The feedback around the current implementation in the 'Miscellaneous' section is around renaming and minor refactoring of code. I agree with most of the feedback and some choices are personal preferences which should discuss and resolve. We should address it soon when we implement the better `Impurity` interface that we promised to implement ASAP. I think the "Error" interface you described is very similar to what @mengxr proposed as well. We should discuss the naming convention. Even though I don't feel strongly about "Impurity" but "Error" might not be the best name for the classification scenario. I am open to better names and ready to be convinced otherwise. :-) For 'General Design Notes', I have similar thoughts but I will wait for @etrain's comments since he has thought about it carefully. In general, I like @etrain 's MLI design for Model and Algorithm. I did not tie the current implementation to the existing traits yet since I wanted to have a broader conversation about it after the tree PR. It's straightforward to implement once we agree on the interfaces for mllib algorithms. Finally, and most importantly, thanks a ton for performing such extensive tests on a massive dataset! The results are not too shabby. ;-)

github_pulls_reviews:

1. Organize imports and remove unused ones.
2. Use JavaDoc style.
3. Remove extra blank line.
4. No more than 100 chars per line.
5. Use 2-space indentation.
6. no space between "input" and ":"
7. put a space between arguments
8. new line at the end
9. put an empty line after header
10. JavaDoc for public methods.
11. Style nit, could add the method header comment using the JavaDoc style like: `` `/** * Method to train ... * ``
12. scala imports should be in a separate group, before spark imports
13. Put an extra space after "///".
14. In the current implementation of other algorithms in MLlib, we let users to choose whether the data should be cached or not. How many passes does your algorithm need?
15. use RDD.first() here
16. Why fixing the seed here? It means certain item won't get selected.
17. extra spaces around "/"
18. remove the extra space after "="
19. put an extra space after ","
20. put an extra space between ")" and "{"
21. no infix
22. extra space between "key" and ":"
23. remove extra empty line
24. What does "DecisionTreeRunner<master>[slices]" mean exactly?
25. isSwitch is not used.
26. using named argument doesn't help improve readability here
27. Does accuracy only apply to classification? If so, we should not output accuracy score for regression.

28. Same question here.
29. Doesn't need to define this var because you calculate the value in one line and the method name tells what it is.
30. Same question was asked by @srowen : It is easy to support multi-class? Also, why 0.5 is used here as the threshold?
31. use `"/*** ... */"`
32. Because this is called for many times, we should use binary search or at least a while loop instead of for.
33. This comment is not clear to me.
34. need an extra space for indentation
35. extra empty line
36. why do you create a for loop and break at the first iteration?
37. replace the for loop by a while loop
38. remove this comment
39. Use JavaDoc style.
40. There are several nested methods defined inside `findBestSplits`. Some of them are complex enough to have unit tests of their own.
41. Define a constant for `"-1"` for better readability.
42. `"arr(validSignalIndex) != -1"` should be sufficient here
43. remove `"if .. else"`
44. use `"nodeIndex"` or simply `"i"` instead of `"node"`
45. Is it necessary to fill in the rest with dummy splits? Consider the case when we have one feature with 1000 categories and all other features are binary.
46. Agree. Was planning to add it but forgot about it while trying to write a working version. I will update this when I add tests for this method.
47. I will add more details to this. This is the key component to understanding the optimizations and is not simple to understand (and hence requires more docs).
48. 0.5 was just a threshold used for verification. I will make it configurable and for now return a double value between 0 and 1 similar to other classification algorithms in `mllib`. This will make it easier for performing ROC/AUC calculations.
49. That's a great observation. It will also need a corresponding change in the `findBin` calculation.
50. should this be private?
51. private?
52. might make sense to make this private as well
53. remove unused import
54. @pwendell I think you are pointing to the `findSplitsBins` method. Yes, it can be private. I will make the change.
55. Sure.
56. @pwendell This `train` method could be called externally especially if you have categorical features. I think keeping this method public is fine.
57. Need doc for public members.
58. @mengxr Sorry, it's not clear to me what you meant. Could you please elaborate?
59. If you want to leave it public, then you need to add JavaDoc for it.
60. This method already has a javadoc.
61. Sorry for the confusion. Both @pwendell and I were talking about ``` val InvalidBinIndex = -1 ``` instead of ``train``.
62. This is the wiki page for ``Gini coefficient``, which is different from ``Gini impurity``. Should change to http://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity
63. Should use ``java.lang.UnsupportedOperationException``.
64. Should use ``java.lang.UnsupportedOperationException`` and organize imports.
65. It is easy to loss precision or run into overflow in the computation of ``sumSquares``. Is it only for computing the sample variance? If true, we can simplify this interface to accept variance directly. We have a more stable implementation of variance computation in ``DoubleRDDFunctions``.
66. That's a nice observation. However, using the ``variance`` calculation in ``StatCounter`` might be slow since the `merge` method recomputes ``n, mu, m2`` for each value. Also, it won't fit well with the ``binCombOp`` operation in the ``aggregate`` function. One can probably optimize the ``def merge(values: TraversableOnce[Double])`` `StatCounter`` method in the ``Variance`` class by doing a batch or mini-batch update for both speed and precision but that's a separate discussion. I see your concern with computing ``sumSquares`` for a large fraction of the instances and I think it's best to leverage the ``def merge(other: StatCounter): StatCounter`` method. We can calculate `StatCounter` per partition using ``count, sum, sumSquares`` and then merge during ``binCombOp`` for numerical stability. I won't be hard to implement. Let me know what you think.
67. I agree with Manish. Numerical stability is the first thing that comes to mind on seeing a large ``avg = sum/count`` calculation. In practice, I haven't seen any significant difference in results or overflows with even billion sample datasets. Also, features in machine learning are typically normalized and dynamic range is small (bounded away from 0 and infinity). We definitely cannot use the methods in `DoubleRDDFunctions` because we want to calculate the variance of various splits, which requires the stats to be "aggregable". But we may be able to modify the api's

- to use (count, avg, avgSquares) as the stats and make the calculations more stable. E.g., to merge (count, avg) of two parts `(c1, a1)`, `(c2, a2)`, we would have `(c1 + c2, a1 * (c1/(c1+c2)) + a2 * (c2/(c1+c2)))`. Not too keen on that change, but let me know if that works.
68. I agree that overflow is an issue here (particularly in the case of sumSquares), but also agree with Manish/Hirakendu that this algorithm maintains its ability to generate a tree in a reasonable amount of time based on this property that we compute statistics for splits and then merge them together. I actually do think it makes sense to maintain "(count, average, averageSumSq)" for each partition in a way that's overflow friendly and compute the combination as count-weighted average of both as Hirakendu suggests. This will complicate the code but should solve the overflow problem and keep things pretty efficient. That said - maybe this could be taken care of in a future PR as a bugfix, rather than in this one?
 69. The major loss of precision is from `sumSquares - sum*sum/count`, where a large number subtracts another. Changing the interface to `(count, avg, avgSquare)` would help avoid overflow but has nothing to do with precision. I agree with @etrain that we can improve it in a future PR. The question is whether we should make `calculate(c0, c1)` and `calculate(count, sum, square)` a public method of `Impurity`. In either classification or regression, `Impurity` works like an accumulator. What we need is to describe how to process a label of type Double, how to merge two `Impurity` instances, and how to get impurity from an instance, which is very similar to `StatCounter`. It is strange to see Gini only implements the first but not the second, while Variance only implements the second but not the first. We probably need to reconsider the design here. For example, if we want to handle three classes in the future, we will run into a signature collision with `calculate(count, sum, squareSum)`.
 70. I agree with @mengxr that the design needs to be made more generic for extensibility -- the signature collision is an issue we will encounter soon. As @mengxr mentioned, the impurity class should specify three methods: a) a method to calculate impurity (or an intermediate value) for a single label, b) the `binSeqOp` method and c) the `binCombOp` method. We will encounter a loss in performance if we calculate the stats per instance and then "merge" them and a loss in precision (as @mengxr pointed out) if we calculate stats after accumulating large numbers. I wonder whether the sweet spot is to calculating the stats (impurity) per partition without encountering a significant loss in performance and precision. The important question is that whether this issue "blocks" this PR or belongs to a separate PR. I vote for the postponing it to a future PR. We will soon encounter more loss functions during ensemble implementations (GBT for example) so it might be good to handle it then.
 71. @manishamde As you mentioned, we can do partition-wise accumulation and then merge them without performance loss. It is okay to internally accumulate `sum` and `sumSquare` in `Variance` and use them to compute variance in this PR. But I think it is necessary to update the `Impurity` interface in this PR to make it more general, because it is a public trait.
 72. @mengxr How about a simple temporary fix where we just have one `calculate` method in the `Impurity` trait that takes a list of Double as a single parameter? I plan to write a generic implementation (as discussed) soon but it might take me a few more days and will delay the PR.
 73. @manishamde Do you need to merge two `Impurity` instances (from two partitions) then?
 74. @mengxr Currently, the `calculate` method is used on bin aggregated data after the `aggregate` operation. So this change won't require doing partition-wise merge of impurities. It's just code restructuring without modifying the implementation.
 75. @mengxr Are we in agreement here for the immediate fix for the PR? I think we all agree on the final version but wanted to make sure we have consensus for this PR before I implement your suggestion.
 76. @manishamde I prefer `accumulate(Double)` and `get(): Double`. But if a single `calculate(Seq[Double])` won't run into memory issues, I'm okay with the change. Sorry for my late response!
 77. @mengxr I am not clear what you meant by the `accumulate` and `get` methods. There is no problem in encountering memory issues with `calculate(Seq[Double])`. The length of the input sequence will be 2 for classification implementations (Gini, Entropy) and 3 for regression implementation (Variance). The `calculate` method is only used after the bin aggregation and subsequent bin aggregate to split aggregate conversions. It's just a minor code change to have a single method signature in the `Impurity` trait.
 78. @manishamde Sorry I misunderstood your proposal. I thought `Seq[Double]` contains the raw labels, like `` 0 1 1 0 1 0 0 1 `` for classification and `` 1.0 2.0 -3.0 5.0 `` for regression. You meant that `Seq[Double]` is `Seq(c0, c1)` for classification and `Seq(n, sum, sumSq)` for regression. I can see this would be a minor code change but it makes the interface more confusing, IMHO. If my understanding is correct, Impurity is applied to a sequence of labels. The code will have better separation if we use the following:

```
labels.foreach(impurityAccumulator.accumulate) impurityAccumulator += anotherImpurityAccumulator val impurity = impurityAccumulator.get
```

I think this is also a minor code change and it gives a cleaner public interface of `Impurity`. Maybe we can get some suggestions from others. @mateiz ?
 79. I'm just catching up on this, but is the problem that there will be other types of Impurity later that calculate different stats (not just variance)? In that case, maybe we can have Impurity be parameterized (Impurity[T]) where T is a type it accumulates over. However I'd also be okay with leaving this as is initially and marking the API unstable if this is an internal API. The question is how many users will call this directly.
 80. BTW I'd also be okay updating this API in a later pull request before we release 1.0. It's fair game to change new APIs in that time window.

81. @mateiz A user needs an ``Impurity`` instance to construct ``Strategy``, but very unlikely they need to call ``calculate`` directly or implement their own ``Impurity``. I'm okay if we mark the ``calculate`` method unstable in another PR later.
82. @mengxr The generic interface you noted is correct. However, I think implementing this generic interface and the corresponding implementations is not a minor code change. There are some assumptions in the bin aggregation code that may need to be updated and it also requires adding partition-wise impurity calculation and aggregation. @mateiz As @mengxr noted, it's highly unlikely that a user will write their own ``Impurity`` implementation. It's mostly an internal API and could be addressed soon in a different PR. I think we all agree (please correct me if I am wrong) the ``Impurity`` update belongs to a different PR. I can spend time on it immediately after this PR is accepted. Is this the correct method of marking a method as unstable using the javadoc? `ALPHA COMPONENT`
83. Need docs on ``numBins`` and explain what it means if it is set manually as in the test suite.
84. Need comments either here or in the JavaDoc (preferred) on what this line does.
85. Will do. Thanks!
86. Will do. Thanks!
87. Adding to the discussion on the need for a generic interface for ``Impurity``, or more precisely ``Error``, I believe we all see that it's good to have. Ideally I would have preferred a single ``Error`` trait and that all types of Error like Square or KL divergence extend it, but the consensus is that it negatively impacts performance. In addition to performance-oriented implementations for specific loss functions, I would still recommend a generic ``Error`` interface and a generic implementation of decision-tree based on this interface. One possibility is to add a third ``calculate(stats)``, or more precisely ``error(errorStats: ErrorStats)`` to the ``Error`` interface. I am not sure it will help the signature collision problem though, unless we just keep the one signature for generic error statistics. For reference and example of one such interface and implementations, see ``trait LossStats[S <: LossStats[S]]`` and ``abstract class Loss[S <: LossStats[S]:Manifest]`` in my previous PR, <https://github.com/apache/incubator-spark/pull/161/files>, that exactly do that and provide interfaces for aggregable error statistics and calculating error from these statistics. (On second thought, I feel ``ErrorStats`` and ``Error`` are better names.) Also see the generic implementation ``class DecisionTreeAlgorithm[S <: LossStats[S]:Manifest]`` and implementations of specific error functions, ``SquareLoss`` and ``EntropyLoss``.
88. ``DecisionTree`` class and object should be reorganized and separated into ``DecisionTreeModel`` and ``DecisionTreeAlgorithm``, with a ``Strategy`` and root ``Node`` as part of ``Model`` and ``train`` as part of the ``Algorithm``.
89. ``Filter`` class is a nice abstraction of branching conditions leading to current node. There are already references to left and right child nodes, so I think this is redundant. If need be, a reference to a parent node as an ``Option[Node]`` suffices and is more useful. The functionality should be covered across ``Node`` and ``Split`` classes.
90. The functionality of ``Split`` can be simplified by a modification. If I understand correctly, ``Split`` represents the left or right (low or high) branch of the parent node. Instead, it suffices to store the branching condition for each node as a splitting condition. This can be appropriately named as ``SplitPredicate`` or ``SplittingCondition`` or branching condition and consist of feature id, feature type (continuous or categorical), threshold, left branch categories and right branch categories. I think depending on the choice here, we require ``Filter``, but nonetheless I think it's redundant and we should exploit the recursive/linked structure of tree, which we are doing anyway.
91. ``Node`` should be a recursive/linked structure with references to child nodes and parent node (the latter allows for easy traversal), so I don't see the need for ``nodes: Array[Node]`` as the model and the ``build`` method in ``Node``. The ``DecisionTreeModel`` should essentially be the root ``Node`` with methods like ``predict`` etc. involving a recursive traversal on child nodes. The method ``predictIfleaf`` in ``Node`` should be renamed to simply ``predict``. It predicts regardless of whether it's a leaf and does recursive traversal until it hits a leaf child. The prediction value should be renamed to ``prediction`` instead of ``predict``, which would clean up the ambiguity with this ``predict`` method. Putting things together: ``Node`` should be simple with a ``prediction`` and should be a recursive structure. ``DecisionTreeModel`` should be a wrapper around a root ``Node`` member and contain methods like ``predict``, ``save``, ``explain``, ``load`` etc. based on recursive traversal.
92. ``Strategy`` should be renamed to ``Parameters``. Modeling and algorithm parameters can be separate, the latter being part of the model.
93. ``Bin`` class can be simplified and some members renamed. The ``lowSplit``, ``highSplit`` can be simplified to the single threshold corresponding to the left end of the bin range. This can be named to ``leftEnd`` or ``lowEnd``. It's not clear this class is needed at first place. For categorical variables, the value itself is the bin index, and for continuous variables, bins are simply defined by candidate thresholds, in turn defined by quantiles. For every feature id, one can maintain a list of categories and thresholds and be done. In that case, for continuous features, the position of the threshold is the bin index.
94. ``InformationGainStats`` and ``Split`` nicely separate the members of ``Node``, but can also be flattened and put at top level. Would make storage and explanation slightly easier, albeit less unstructured.
95. ``Impurity`` should be renamed to ``Error`` or something more technical and familiar. Also see the comments earlier for the necessity and example design of a generic ``Error`` interface. The ``calculate`` method can be renamed to

- something verbose like `error`. For a generic interface, an additional `ErrorStats` trait and `error(errorStats: ErrorStats)` method can be added. For example, `Variance` or more aptly, `SquareError`, would implement `case class SquareErrorStats(count: Long, mean: Double, meanSquare: Double)` and `error(errorStats) = errorStats.meanSquare - errorStats.mean * errorStats.mean / count`. Note that `ErrorStats` should have aggregation methods, e.g., it's easy to see the implementation for `SquareErrorStats`. The `Variance` class should be renamed to `SquareError`, `Entropy` to `EntropyError` or `KLDivergence`, `Gini` to `GiniError`.
96. The `Algorithm` Enumeration seems redundant given `Impurity` which implies the `Algorithm` anyway.
 97. The various `Enumeration` classes in `mllib.tree.configuration` package are neat. A uniform `_design pattern_` for parameters and options should be used for MLLib and Spark, and this could be a start. Alternatively, if there is an existing pattern in use, it should be followed for decision tree as well.
 98. Input data formats for main program should be made consistent with programs for other algorithms. Currently, it's CSV for decision-tree, but I believe it is `>,<feature1>\t<feature2>...` etc for other algorithms. I prefer a more `_standard_` TSV format as used in Hadoop text format. Alternatively, if the label has to be separated, current `_labeled point_` format used in other programs is fine.
 99. A plan should be made to have a consistent hierarchy and organization of various algorithms in the MLLib package. A separate ``tree`` subpackage seems unnecessary.

jira_issues:

1. **summary:** Shutdown the SessionManager timeoutChecker thread properly upon shutdown
description: Shutdown for SessionManager waits 10seconds for all threads on the threadpoolExecutor to shutdown correctly. The cleaner thread - with default settings - will take 6 hours to shutdown, so essentially any shutdown of HS2 is always delayed by 10s. The cleaner thread should be shutdown properly.

jira_issues_comments:

1. [~thejas], [~prasanth_j] - please review. This cuts the test runtime of TestXSRFDFilter by 40 seconds, and likely other tests as well.
2. Here are the results of testing the latest attachment:
<https://issues.apache.org/jira/secure/attachment/12829914/HIVE-14817.01.patch> {color:red}ERROR:{color} -1 due to no test(s) being added or modified. {color:red}ERROR:{color} -1 due to 6 failed/errored test(s), 10555 tests executed *Failed tests:* {noformat} org.apache.hadoop.hive.cli.TestCliDriver.testCliDriver[acid_mapjoin] org.apache.hadoop.hive.cli.TestCliDriver.testCliDriver[ctas] org.apache.hadoop.hive.cli.TestCliDriver.testCliDriver[vector_join_part_col_char] org.apache.hadoop.hive.cli.TestMiniTezCliDriver.testCliDriver[explainuser_3] org.apache.hadoop.hive.metastore.TestMetaStoreMetrics.testMetaDataCounts org.apache.hive.jdbc.TestJdbcWithMiniHS2.testAddJarConstructorUnCaching {noformat} Test results: <https://builds.apache.org/job/PreCommit-HIVE-Build/1278/testReport> Console output: <https://builds.apache.org/job/PreCommit-HIVE-Build/1278/console> Test logs: <http://ec2-204-236-174-241.us-west-1.compute.amazonaws.com/logs/PreCommit-HIVE-Build-1278/> Messages: {noformat} Executing org.apache.hive.ptest.execution.TestCheckPhase Executing org.apache.hive.ptest.execution.PrePhase Executing org.apache.hive.ptest.execution.ExecutionPhase Executing org.apache.hive.ptest.execution.ReportingPhase Tests exited with: TestsFailedException: 6 tests failed {noformat} This message is automatically generated.
ATTACHMENT ID: 12829914 - PreCommit-HIVE-Build
3. +1
4. Committed. Thanks for the review.