Common Code

This notebook contains many of the functions the other notebooks use. Consider exporting it into your own notebooks to help you in your initial explorations. Pull-requests to improve it are welcome.

Getting public datasets

Mathematica makes it easy to access public datasets. These functions are light wrappers around its powerful built-in functionality.

In[1]:=
```
Clear[ImportPennMLBenchmarksDataset]
ImportPennMLBenchmarksDataset[datasetName_String] :=
 Module[{filename, tsvHeader, benchmarkData}, filename =
   "https://github.com/EpistasisLab/penn-ml-benchmarks/blob/master/datasets/" <>
    datasetName <> "/" <> datasetName <> ".tsv.gz?raw=true";
  tsvHeader = Import[filename, "TSV"] // First;
  benchmarkData =
   Import[filename, "TSV"] // Rest // GroupBy[#, Last] & // Map[Most, #, {2}] &;
  {tsvHeader, benchmarkData}]
```

Creating feature splits

In[3]:=
```
Clear[FeaturePartition]
FeaturePartition[features_List,
  typeAndCount : {{_String, _Integer} ...}, n_Integer] :=
 Module[{type, count},
  GroupBy[features, Last] //
    Table[{type, count} = tAndC //
        RandomSample[#[type], count], {tAndC, typeAndCount}] & //
   Partition[#, n] &]
```

In[5]:=
```
uciAdultFeatures =
 {{1, "Numerical"}, {2, "Nominal"}, {3, "Numerical"}, {4, "Nominal"}, {5, "Nominal"},
  {6, "Nominal"}, {7, "Nominal"}, {8, "Nominal"}, {9, "Nominal"}, {10, "Nominal"},
  {11, "Numerical"}, {12, "Numerical"}, {13, "Numerical"}, {14, "Nominal"}}
```

Out[5]=
```
{{1, Numerical}, {2, Nominal}, {3, Numerical}, {4, Nominal}, {5, Nominal},
 {6, Nominal}, {7, Nominal}, {8, Nominal}, {9, Nominal}, {10, Nominal},
 {11, Numerical}, {12, Numerical}, {13, Numerical}, {14, Nominal}}
```

Training

In[6]:=
```
Clear[FilterTraining]
FilterTraining[data_, {alphaIndices_, betaIndices_}] :=
 Association[0 → data[0][[alphaIndices]], 1 → data[1][[betaIndices]]]
```

In[8]:=
```
Clear[TrainClassifiersDisjoint]
TrainClassifiersDisjoint[classifiersData_,
  classifierTypes_List, trainIndices_List, featureTypes_List] := Module[
  { trainingSamples, classifiers},
  classifiers = Transpose@{
      classifierTypes,
      Transpose@{Map[First, classifiersData, {2}], trainIndices} //
       Map[FilterTraining @@ # &, #] &,
      featureTypes} //
    Map[Classify[#[[2]], Method → #[[1]], TrainingProgressReporting → None,
        PerformanceGoal → "DirectTraining", FeatureTypes → #[[3]] ] &, #] &;
  classifiers]
```

Testing

All benchmarking of algebraic evaluators requires that we know the ground truth. The basic data structure for carrying out the investigations is to keep separate the vote counts for the ensemble by true label. These functions help us construct
it.

In[10]:=
```
Clear[LabelVotingCounts]
LabelVotingCounts[classifiers_, classifiersData_] := Module[
  {nTestAlpha, nTestBeta, decisions,
   alphaSamples, betaSamples,
   byLabelDecisions, votingPatternCountsByLabel,
   sols, equationsToSolve, vars,
   gt, alphaLabel = 0, betaLabel = 1},
  (* Calculate the size of the test sets *)
  {nTestAlpha, nTestBeta} = classifiersData // First // Map[Length, #, {2}] & //
     {#[alphaLabel], #[betaLabel]} & // Last /@ # &;
  (* We arbitrarily define "0" as the alpha label, and "1" as the beta label *)
  decisions = Table[classifiers[[i]][classifiersData[[i]] // Map[Last, #] & //
      Join[#[alphaLabel], #[betaLabel]] &], {i, Length@classifiers}];
  alphaSamples = RandomSample[Range@nTestAlpha, nTestAlpha];
  betaSamples = RandomSample[Range@nTestBeta, nTestBeta];
  byLabelDecisions =
   decisions // Map[TakeDrop[#, nTestAlpha] &, #] & // Map[Association[
        {alphaLabel → #[[1]][[alphaSamples]], betaLabel → #[[2]][[betaSamples]]}] &, #] &;
  votingPatternCountsByLabel = byLabelDecisions // Merge[#, Identity] & //
     Map[Transpose, #] & // Map[Counts, #] &;
  votingPatternCountsByLabel
 ]
```

In[12]:=
```
Clear[VotingFrequenciesData]
VotingFrequenciesData[testAlignedDecisions_Association,
  classifiers : {_Integer ..}] := Module[
  {eventCounts},
  eventCounts =
   Values@testAlignedDecisions // Merge[#, Identity] & // Map[Total, #] & //
         KeyValueMap[(#1[[classifiers]] → #2) &, #] & //
        GroupBy[#, First] & //
       Map[Last, #, {2}] & //
      Map[Total, #] & //
     KeyMap[Subscript[f, Sequence @@ Map[If[# == 0, α, β] &, #]] &, #] & //
    # / Total@# &]
```

In[14]:=
```
Clear[TurnVotesToIndicators]
TurnVotesToIndicators[key_, label_] := Map[If[# === label, 1, 0] &, key]

Clear[ClassifierLabelAccuracy]
ClassifierLabelAccuracy[
  voteCountsByLabel_Association, classifier_Integer, label_] :=
 voteCountsByLabel[label] // KeyMap[TurnVotesToIndicators[#, label] &, #] & //
      Normal // GroupBy[#, #[[1, classifier]] &] & //
    Map[Last, #, {2}] & // Map[Total, #] & //
  #[1] / (Total@#) &
```

In[18]:=
```
Clear[ProjectVoteCounts]
ProjectVoteCounts[labelVoteCounts_, classifiers_List] :=
 Normal[labelVoteCounts] // GroupBy[#, #[[1, classifiers]] &] & //
   Map[Last, #, {2}] & // Map[Total, #] &
```

Functions for correlations

In[20]:=
```
Clear[CorrelationProduct]
CorrelationProduct[indicators_List, accuracies_List] :=
 Times @@ (indicators - accuracies)

Clear[LabelCorrelations]
LabelCorrelations[voteCountsByLabel_Association,
  classifiers_List, label_] := Module[
    {labelAccuracies},
    labelAccuracies =
     Map[ClassifierLabelAccuracy[voteCountsByLabel, #, label] &, classifiers];
    voteCountsByLabel[label] // ProjectVoteCounts[#, classifiers] & //
      KeyMap[TurnVotesToIndicators[#, label] &, #] & //
     KeyMap[CorrelationProduct[#, labelAccuracies] &, #] & ] // Normal //
  ((Map[Times @@ # &, #] // Total) / (Map[Last, #] // Total)) &
```

Algebraic evaluation of three error independent binary classifiers

In[24]:=
```
Clear[MakeIndependentVotingIdeal]
MakeIndependentVotingIdeal[{i_, j_, k_}] :=
 {P_α P_i,α P_j,α P_k,α + (1 - P_α) (1 - P_i,β) (1 - P_j,β) (1 - P_k,β) - f_α,α,α,
  P_α P_i,α P_j,α (1 - P_k,α) + (1 - P_α) (1 - P_i,β) (1 - P_j,β) P_k,β - f_α,α,β,
  P_α P_i,α (1 - P_j,α) P_k,α + (1 - P_α) (1 - P_i,β) P_j,β (1 - P_k,β) - f_α,β,α,
  P_α P_i,α (1 - P_j,α) (1 - P_k,α) + (1 - P_α) (1 - P_i,β) P_j,β P_k,β - f_α,β,β,
  P_α (1 - P_i,α) P_j,α P_k,α + (1 - P_α) P_i,β (1 - P_j,β) (1 - P_k,β) - f_β,α,α,
  P_α (1 - P_i,α) P_j,α (1 - P_k,α) + (1 - P_α) P_i,β (1 - P_j,β) P_k,β - f_β,α,β,
  P_α (1 - P_i,α) (1 - P_j,α) P_k,α + (1 - P_α) P_i,β P_j,β (1 - P_k,β) - f_β,β,α,
  P_α (1 - P_i,α) (1 - P_j,α) (1 - P_k,α) + (1 - P_α) P_i,β P_j,β P_k,β - f_β,β,β}
```

In[26]:=
```
Clear[AlgebraicallyEvaluateClassifiers]
AlgebraicallyEvaluateClassifiers[classifiers_, classifiersData_] := Module[
  { votingPatternCountsByLabel, evaluationIdeal,
   equationsToSolve, vars, sols, gt, observableFrequencies},
  (* Calculate the size of the test sets *)
  votingPatternCountsByLabel = LabelCounts[classifiers, classifiersData];
  sols = Table[
    observableFrequencies = VotingFrequenciesData[votingPatternCountsByLabel, trio];
    evaluationIdeal = MakeIndependentVotingIdeal[trio];
    equationsToSolve = Map[(# == 0) &, evaluationIdeal] /. observableFrequencies;
    vars = Variables /@ evaluationIdeal // Flatten // DeleteDuplicates //
        SortBy[#, {Last@#} &] & // Cases[#, Except[f__]] &;
    Map[SortBy[#, Last@First@# &] &, Solve[equationsToSolve, Reverse@vars]],
    {trio, Subsets[Range@Length@classifiers, {3}]}];
  gt = GTClassifiers[votingPatternCountsByLabel];
  {gt, sols}
 ]
```

In[28]:=
```
Clear[GBIndependentSystem]
GBIndependentSystem[classifiers_, classifiersData_] := Module[
  { votingPatternCountsByLabel, evaluationIdeal,
   equationsToSolve, vars, sols, gt, observableFrequencies},
  (* Calculate the size of the test sets *)
  votingPatternCountsByLabel = LabelCounts[classifiers, classifiersData];
  sols = Table[
    observableFrequencies = VotingFrequenciesData[votingPatternCountsByLabel, trio];
    evaluationIdeal = MakeIndependentVotingIdeal[trio];
    equationsToSolve = Map[(#) &, evaluationIdeal] /. observableFrequencies;
    vars = Variables /@ evaluationIdeal // Flatten // DeleteDuplicates //
       SortBy[#, {Last@#} &] & // Cases[#, Except[f__]] &;
    Print[vars];
    GroebnerBasis[equationsToSolve, Reverse@vars],
    {trio, Subsets[Range@Length@classifiers, {3}]}];
  gt = GTClassifiers[votingPatternCountsByLabel];
  {gt, sols}
 ]
```

Measuring the error correlation of binary classifiers

In[30]:=
```
Clear[LabelRMSE]
LabelRMSE[gtDiff_, label_] :=
 KeySelect[gtDiff, (Length@# == 3) &] // KeySelect[#, (Last@# == label) &] & // Values //
    Map[#^2 &, #] & // Mean // Sqrt

Clear[GTClassifiers]
GTClassifiers[votingPatternCountsByLabel_Association] := Module[
  {alphaLabel = 0, nClassifiers},
  nClassifiers =
   votingPatternCountsByLabel // First // Keys // RandomChoice // Length;
  Join[{Pα → (votingPatternCountsByLabel // Map[Values, #] & // Map[Total, #] & //
        #[alphaLabel] / Total@# &)},
    Table[Pi,α → ClassifierLabelAccuracy[votingPatternCountsByLabel, i, 0],
     {i, nClassifiers}],
    Table[Pi,β → ClassifierLabelAccuracy[votingPatternCountsByLabel, i, 1],
     {i, nClassifiers}],
    Table[ΓSequence@@pair,α → LabelCorrelations[votingPatternCountsByLabel, pair, 0],
     {pair, Subsets[Range@nClassifiers, {2}]}],
    Table[ΓSequence@@pair,β → LabelCorrelations[votingPatternCountsByLabel, pair, 1],
     {pair, Subsets[Range@nClassifiers, {2}]}],
    Table[ΓSequence@@trio,α → LabelCorrelations[votingPatternCountsByLabel, trio, 0],
     {trio, Subsets[Range@nClassifiers, {3}]}],
    Table[ΓSequence@@trio,β → LabelCorrelations[votingPatternCountsByLabel, trio, 1],
     {trio, Subsets[Range@nClassifiers, {3}]}]] // Association
 ]
```

In[34]:=
```
Clear[LabelCounts]
LabelCounts[classifiers_, classifiersData_] := Module[
  {nTestAlpha, nTestBeta, decisions,
   byLabelDecisions, votingPatternCountsByLabel,
   sols, equationsToSolve, vars,
   gt, alphaLabel = 0, betaLabel = 1},
  (* Calculate the size of the test sets *)
  {nTestAlpha, nTestBeta} = classifiersData // First // Map[Length, #, {2}] & //
     {#[alphaLabel], #[betaLabel]} & // Last /@ # &;
  (* We arbitrarily define "0" as the alpha label, and "1" as the beta label *)
  decisions = Table[classifiers[[i]][classifiersData[[i]] // Map[Last, #] & //
      Join[#[alphaLabel], #[betaLabel]] &], {i, Length@classifiers}];
  byLabelDecisions = decisions // Map[TakeDrop[#, nTestAlpha] &, #] & //
    Map[Association[{alphaLabel → #[[1]], betaLabel → #[[2]]}] &, #] &;
  votingPatternCountsByLabel = byLabelDecisions // Merge[#, Identity] & //
     Map[Transpose, #] & // Map[Counts, #] &;
  votingPatternCountsByLabel
 ]
```

Code for Experiments

Testing feature partitions

In[36]:=
```
Clear[IsEvaluatorFailureImaginaryQ]
IsEvaluatorFailureImaginaryQ[algebraicSols_List] :=
 (* We only need to test on one of the two solutions*)
 First@algebraicSols //
     (* Extracting the numerical estimates *)
     Last /@ # & //
    (* Are any of them imaginary? *)
    Head /@ # & // Counts //
  If[Lookup[#, Complex, 0] > 0, True, False] &
```

In[38]:=
```
Clear[IsEvaluatorFailureOutOfBoundsRealQ]
IsEvaluatorFailureOutOfBoundsRealQ[sols_List] :=
 If[IsEvaluatorFailureImaginaryQ@sols == False,
  First@sols // Last /@ # & //
   (
     (Min@# < 0) ||
      (Max@# > 1)) &,
  False]
```

In[40]:=
```
Clear[DetectFailedEvaluatorRuns]
DetectFailedEvaluatorRuns[runs_] := Last /@ runs // Flatten // Normal // Last /@ # & //
    (* Turn them into reals *)
    N //
  (* Testing that only real numbers are returned *)
  ((Head /@ # // Counts // Keys // MatchQ[#, {Real}] &) &&
     (* Testing that they are withing [0,1] real bounds *)
     (Min@# > 0) && (Max@# < 1)) &
Clear[IsFailedEvaluatorSols]
IsFailedEvaluatorSols[sols_] := Last@sols // Flatten // Normal // Last /@ # & //
    N //
   ((Head /@ # // Counts // Keys // MatchQ[#, {Real}] &) &&
      (Min@# > 0) && (Max@# < 1)) & //
  Not
```

In[44]:=
```
Clear[GTAndEvaluationFlagForFeaturePartitions]
GTAndEvaluationFlagForFeaturePartitions[
  data_Association, classifierTypes_List, featurePartitions_List,
  nTrain_Integer, nTestAlpha_Integer, nTests_Integer] := Module[
  {currentPartition, ftests, runTrainTestSplit, trainingIndices,
   classifiersData, classifiers, nClassifiers = Length@classifierTypes},
  Map[
   (currentPartition = #;
    ftests = {};
    For[i = 0, i < nTests, i++,
     (* The random train/test split *)
     runTrainTestSplit = Association[
       0 → ({Take[data[0] // First // RandomSample, nTrain],
          Take[data[0] // Last // RandomSample, nTestAlpha]}),
       1 → ({Take[data[1] // First // RandomSample, nTrain],
          Take[data[1] // Last // RandomSample, 2 * nTestAlpha]})];

     trainingIndices = Transpose@{
         RandomSample[Range@nTrain] //
           Partition[#, Floor[nTrain / nClassifiers]] & // Take[#, nClassifiers] &,
         RandomSample[Range@nTrain] //
           Partition[#, Floor[nTrain / nClassifiers]] & // Take[#, nClassifiers] &};

     (* Prepping the data for training and testing *)
     classifiersData = Table[Map[#[[First@Transpose@features]] &,
         runTrainTestSplit, {3}], {features, currentPartition}];

     (* Training *)
     classifiers = TrainClassifiersDisjoint[classifiersData, classifierTypes,
       trainingIndices, Map[(Last@Transpose@#) &, currentPartition]];

     (* Algebraic evaluation *)
     AlgebraicallyEvaluateClassifiers[
         classifiers, classifiersData] // {First@#,
          IsFailedEvaluatorSols@Last@# } & // AppendTo[ftests, #] &];
    ftests) &,
   featurePartitions]]
```

In[46]:=
```
Clear[GTFeaturePartitions]
GTFeaturePartitions[data_Association,
  classifierTypes_List, featurePartitions_List] := Module[
  {currentPartition, runTrainTestSplit, trainingIndices, classifiersData,
   classifiers},
  Map[
   (currentPartition = #;
    Table[
     (* The random train/test split *)
     runTrainTestSplit = Association[
       0 → ({Take[data[0] // First // RandomSample, 6000],
          Take[data[0] // Last // RandomSample, 200]}),
       1 → ({Take[data[1] // First // RandomSample, 6000],
          Take[data[1] // Last // RandomSample, 400]})];

     trainingIndices = Transpose@{
        RandomSample[Range@6000] // Partition[#, 2000] &,
        RandomSample[Range@6000] // Partition[#, 2000] &};

     (* Prepping the data for training and testing *)
     classifiersData = Table[Map[#[[First@Transpose@features]] &,
        runTrainTestSplit, {3}], {features, currentPartition}];

     (* Training *)
     classifiers = TrainClassifiersDisjoint[classifiersData, classifierTypes,
       trainingIndices, Map[(Last@Transpose@#) &, currentPartition]];

     (* Algebraic evaluation *)
     AlgebraicallyEvaluateClassifiers[classifiers, classifiersData] // First,
     {10}]) &,
   featurePartitions]
 ]
```

Universal Surface in Accuracies+Prevalence Space

In[48]:=
```
Clear[GroundTruthImplicitRegion]
GroundTruthImplicitRegion[
  voteCountsByLabel_Association, classifiers : {__Integer}, label_] :=
 Module[{eqs, vars, prevalence, prevalenceTerms},
  eqs = Table[{P_{i,α} - f_α, P_{i,β} - f_β} /.
     VotingFrequenciesData[voteCountsByLabel, {i}], {i, classifiers}];
  vars = Flatten@eqs // Variables /@ # & // Flatten // DeleteDuplicates // Sort;
  prevalence = If[label == α, P_α, P_β];
  prevalenceTerms =
   If[label == α, {prevalence, 1 - prevalence}, {1 - prevalence, prevalence}];
  ImplicitRegion[
   Join[
    (* The single classifier equations *)
    Map[Transpose@{prevalenceTerms, #} &, eqs] //
      Map[Times @@ # &, #, {2}] & //
     Map[(First@# - Last@# == 0) &, #] &,
    (* The pair equations *)
    Table[
     eqs[[pair]] // {First@#, Reverse@Last@#} & // Transpose // Times @@ # & /@ # & //
      (First@# - Last@# == 0) &,
     {pair, Subsets[classifiers, {2}]}]],
   Evaluate@Table[{var, 0, 1}, {var, Join[{prevalence}, vars]}]]]
```