

BudgetedSVM

1.2

Generated by Doxygen 1.8.18

Sun Jun 28 2020 02:22:51

1 BudgetedSVM Documentation	1
2 Hierarchical Index	3
2.1 Class Hierarchy	3
3 Class Index	5
3.1 Class List	5
4 File Index	7
4.1 File List	7
5 Class Documentation	9
5.1 budgetedData Class Reference	9
5.1.1 Detailed Description	11
5.1.2 Constructor & Destructor Documentation	11
5.1.2.1 budgetedData() [1/2]	11
5.1.2.2 budgetedData() [2/2]	12
5.1.3 Member Function Documentation	13
5.1.3.1 distanceBetweenTwoPoints()	13
5.1.3.2 getDataDimensionality()	14
5.1.3.3 getElementOfVector()	14
5.1.3.4 getNumLoadedDataPointsSoFar()	15
5.1.3.5 getSparsity()	16
5.1.3.6 getVectorSqrL2Norm()	16
5.1.3.7 readChunk()	17
5.1.3.8 readChunkAssignments()	19
5.1.3.9 saveAssignment()	19
5.1.4 Member Data Documentation	20
5.1.4.1 assignments	20
5.1.4.2 dimensionHighestSeen	20
5.1.4.3 fAssignFile	21
5.1.4.4 ifileNameAssign	21
5.2 budgetedDataMatlab Class Reference	21
5.2.1 Detailed Description	22
5.2.2 Constructor & Destructor Documentation	22
5.2.2.1 budgetedDataMatlab()	22
5.2.3 Member Function Documentation	23
5.2.3.1 readChunk()	23
5.2.3.2 readDataFromMatlab()	23
5.3 budgetedModel Class Reference	26
5.3.1 Detailed Description	27
5.3.2 Member Function Documentation	27
5.3.2.1 extendDimensionalityOfModel()	27
5.3.2.2 getAlgorithm()	27

5.3.2.3 loadFromFile()	28
5.3.2.4 saveToFile()	29
5.4 budgetedModelAMM Class Reference	30
5.4.1 Detailed Description	30
5.4.2 Member Function Documentation	31
5.4.2.1 extendDimensionalityOfModel()	31
5.4.2.2 getModel()	31
5.4.2.3 loadFromFile()	31
5.4.2.4 saveToFile()	33
5.5 budgetedModelBSGD Class Reference	35
5.5.1 Detailed Description	36
5.5.2 Member Function Documentation	36
5.5.2.1 extendDimensionalityOfModel()	36
5.5.2.2 loadFromFile()	36
5.5.2.3 saveToFile()	38
5.6 budgetedModelLLSVM Class Reference	40
5.6.1 Detailed Description	41
5.6.2 Member Function Documentation	41
5.6.2.1 extendDimensionalityOfModel()	41
5.6.2.2 loadFromFile()	42
5.6.2.3 saveToFile()	44
5.7 budgetedModelMatlab Class Reference	45
5.7.1 Detailed Description	46
5.7.2 Member Function Documentation	46
5.7.2.1 getAlgorithm()	46
5.7.2.2 loadFromMatlabStruct()	47
5.7.2.3 saveToMatlabStruct()	47
5.8 budgetedModelMatlabAMM Class Reference	48
5.8.1 Detailed Description	49
5.8.2 Member Function Documentation	49
5.8.2.1 loadFromMatlabStruct()	49
5.8.2.2 saveToMatlabStruct()	51
5.9 budgetedModelMatlabBSGD Class Reference	54
5.9.1 Detailed Description	54
5.9.2 Member Function Documentation	54
5.9.2.1 loadFromMatlabStruct()	54
5.9.2.2 saveToMatlabStruct()	56
5.10 budgetedModelMatlabLLSVM Class Reference	58
5.10.1 Detailed Description	59
5.10.2 Member Function Documentation	59
5.10.2.1 loadFromMatlabStruct()	59
5.10.2.2 saveToMatlabStruct()	61

5.11 budgetedVector Class Reference	63
5.11.1 Detailed Description	66
5.11.2 Constructor & Destructor Documentation	66
5.11.2.1 budgetedVector()	66
5.11.3 Member Function Documentation	66
5.11.3.1 computeKernel() [1/2]	67
5.11.3.2 computeKernel() [2/2]	67
5.11.3.3 createVectorUsingDataPoint()	68
5.11.3.4 createVectorUsingVector()	69
5.11.3.5 exponentialKernel() [1/2]	69
5.11.3.6 exponentialKernel() [2/2]	70
5.11.3.7 extendDimensionality()	71
5.11.3.8 gaussianKernel() [1/2]	72
5.11.3.9 gaussianKernel() [2/2]	73
5.11.3.10 getDimensionality()	74
5.11.3.11 getID()	74
5.11.3.12 getSqrL2norm()	74
5.11.3.13 linearKernel() [1/2]	74
5.11.3.14 linearKernel() [2/2]	75
5.11.3.15 operator[]() [1/2]	76
5.11.3.16 operator[]() [2/2]	77
5.11.3.17 polyKernel() [1/2]	77
5.11.3.18 polyKernel() [2/2]	78
5.11.3.19 setSqrL2norm()	79
5.11.3.20 sigmoidKernel() [1/2]	79
5.11.3.21 sigmoidKernel() [2/2]	79
5.11.3.22 sqrNorm()	80
5.11.3.23 userDefinedKernel() [1/2]	80
5.11.3.24 userDefinedKernel() [2/2]	81
5.11.4 Member Data Documentation	82
5.11.4.1 array	82
5.11.4.2 arrayLength	82
5.11.4.3 chunkWeight	83
5.11.4.4 id	83
5.11.4.5 sqrL2norm	83
5.11.4.6 weightID	84
5.12 budgetedVectorAMM Class Reference	84
5.12.1 Detailed Description	85
5.12.2 Constructor & Destructor Documentation	85
5.12.2.1 budgetedVectorAMM()	85
5.12.3 Member Function Documentation	86
5.12.3.1 createVectorUsingDataPoint()	86

5.12.3.2 createVectorUsingVector()	86
5.12.3.3 downgrade()	87
5.12.3.4 getDegradation()	87
5.12.3.5 getSqrL2norm()	88
5.12.3.6 linearKernel() [1/2]	88
5.12.3.7 linearKernel() [2/2]	88
5.12.3.8 sqrNorm()	89
5.12.3.9 updateDegradation()	89
5.12.3.10 updateUsingDataPoint()	90
5.12.3.11 updateUsingVector()	90
5.12.4 Member Data Documentation	91
5.12.4.1 degradation	92
5.13 budgetedVectorBSGD Class Reference	92
5.13.1 Detailed Description	93
5.13.2 Constructor & Destructor Documentation	93
5.13.2.1 budgetedVectorBSGD()	93
5.13.3 Member Function Documentation	94
5.13.3.1 alphaNorm()	94
5.13.3.2 downgrade()	94
5.13.3.3 getNumClasses()	95
5.13.3.4 updateSV()	95
5.13.4 Member Data Documentation	96
5.13.4.1 alphas	96
5.14 budgetedVectorLLSVM Class Reference	97
5.14.1 Detailed Description	97
5.14.2 Member Function Documentation	97
5.14.2.1 createVectorUsingDataPointMatrix()	97
5.15 parameters Struct Reference	98
5.15.1 Detailed Description	99
5.15.2 Member Function Documentation	100
5.15.2.1 updateVerySparseDataParameter()	100
5.15.3 Member Data Documentation	100
5.15.3.1 BIAS_TERM	101
5.15.3.2 BUDGET_SIZE	101
5.15.3.3 C_PARAM	101
5.15.3.4 CHUNK_SIZE	102
5.15.3.5 CHUNK_WEIGHT	102
5.15.3.6 CLONE_PROBABILITY	102
5.15.3.7 DIMENSION	103
5.15.3.8 K_MEANS_ITERS	103
5.15.3.9 K_PARAM	103
5.15.3.10 KERNEL	104

5.15.3.11 LAMBDA_PARAM	104
5.15.3.12 MAINTENANCE_SAMPLING_STRATEGY	105
5.15.3.13 NUM_EPOCHS	105
5.15.3.14 NUM_SUBEPOCHS	105
5.15.3.15 OUTPUT_SCORES	106
5.15.3.16 VERY_SPARSE_DATA	106
6 File Documentation	107
6.1 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h File Reference	107
6.1.1 Detailed Description	108
6.1.2 Function Documentation	108
6.1.2.1 fakeAnswer()	108
6.1.2.2 parseInputMatlab()	108
6.1.2.3 printErrorStringMatlab()	114
6.1.2.4 printStringMatlab()	115
6.1.2.5 printUsageMatlab()	115
6.2 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.h File Reference	117
6.2.1 Detailed Description	118
6.2.2 Function Documentation	118
6.2.2.1 predictBSGD()	118
6.2.2.2 trainBSGD()	120
6.3 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.h File Reference	124
6.3.1 Detailed Description	126
6.3.2 Function Documentation	126
6.3.2.1 fgetWord()	126
6.3.2.2 parseInputPrompt()	127
6.3.2.3 printUsagePrompt()	133
6.3.2.4 readableFileExists()	135
6.3.2.5 setPrintErrorStringFunction()	136
6.3.2.6 setPrintStringFunction()	136
6.3.2.7 svmPrintErrorString()	137
6.3.2.8 svmPrintString()	137
6.4 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/llsvm.h File Reference	138
6.4.1 Detailed Description	138
6.4.2 Function Documentation	138
6.4.2.1 predictLLSVM()	138
6.4.2.2 trainLLSVM()	140
6.5 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.h File Reference	142
6.5.1 Detailed Description	143
6.5.2 Function Documentation	143
6.5.2.1 predictAMM()	143
6.5.2.2 trainAMMbatch()	145

6.5.2.3 trainAMMonline()	152
6.5.2.4 trainPegasos()	156
Index	159

Chapter 1

BudgetedSVM Documentation

Thank you for using [BudgetedSVM](#), a toolbox for training large-scale, non-linear classifiers. The toolbox implements the following four SVM/SVM-like algorithms for large-scale, non-linear classification:

- Pegasos (Shalev-Shwartz, S., Singer, Y., Srebro, N., "Pegasos: Primal Estimated sub-GrAdient SOLver for SVM", ICML, 2007)
- AMM batch and AMM online (Wang, Z., Djuric, N., Crammer, K., Vucetic, S., "Trading Representability for Scalability: Adaptive Multi-Hyperplane Machine for Nonlinear Classification", KDD, 2011)
- GAMM (Djuric, N., Wang, Z., Vucetic, S., "Growing Adaptive Multi-hyperplane Machines", ICML 2020)
- BSGD (Wang, Z., Crammer, K., Vucetic, S., "Breaking the Curse of Kernelization: Budgeted Stochastic Gradient Descent for Large-Scale SVM Training", JMLR, 2012)
- LLSVM (Zhang, K., Lan, L., Wang, Z., and Moerchen, F., "Scaling up Kernel SVM on Limited Resources: A Low-rank Linearization Approach", AISTATS, 2012)

Please report any comments/bugs/praises to nemanja@temple.edu. We hope you will find this toolbox useful!

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

budgetedData	9
budgetedDataMatlab	21
budgetedModel	26
budgetedModelAMM	30
budgetedModelMatlabAMM	48
budgetedModelBSGD	35
budgetedModelMatlabBSGD	54
budgetedModelLLSVM	40
budgetedModelMatlabLLSVM	58
budgetedModelMatlab	45
budgetedModelMatlabAMM	48
budgetedModelMatlabBSGD	54
budgetedModelMatlabLLSVM	58
budgetedVector	63
budgetedVectorAMM	84
budgetedVectorBSGD	92
budgetedVectorLLSVM	97
parameters	98

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

budgetedData	Class which handles manipulation of large data sets that cannot be fully loaded to memory (using a data structure similar to Matlab's sparse matrix structure)	9
budgetedDataMatlab	Class which manipulates sparse array of vectors (similarly to Matlab sparse matrix structure), with added functionality to load data directly from Matlab	21
budgetedModel	Interface which defines methods to load model from and save model to text file	26
budgetedModelAMM	Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to text file	30
budgetedModelBSGD	Class which holds the BSGD model (comprising the support vectors stored as budgetedVectorBSGD), and implements methods to load BSGD model from and save BSGD model to text file	35
budgetedModelLLSVM	Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to text file	40
budgetedModelMatlab	Interface which defines methods to load model from and save model to Matlab environment	45
budgetedModelMatlabAMM	Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to Matlab environment	48
budgetedModelMatlabBSGD	Class which holds the BSGD model, and implements methods to load BSGD model from and save BSGD model to Matlab environment	54
budgetedModelMatlabLLSVM	Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to Matlab environment	58
budgetedVector	Class which handles high-dimensional vectors	63
budgetedVectorAMM	Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for AMM algorithms	84
budgetedVectorBSGD	Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for BSGD algorithm	92

[budgetedVectorLLSVM](#)

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for LLSVM algorithm 97

[parameters](#)

Structure holds the parameters of the implemented algorithms 98

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/ budgetedSVM_matlab.cpp . .	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/ budgetedSVM_matlab.h Implements classes and functions used for training and testing of budgetedSVM algorithms in Matlab	107
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/ budgetedsvm_predict.cpp . .	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/ budgetedsvm_train.cpp . . .	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/ libsvmread.c	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/ libsvmwrite.c	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ bsgd.cpp	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ bsgd.h Defines classes and functions used for training and testing of BSGD (Budgeted Stochastic Gra- dient Descent) algorithm	117
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ budgetedsvm-predict.cpp	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ budgetedsvm-train.cpp	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ budgetedSVM.cpp	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ budgetedSVM.h Header file defining classes and functions used throughout the budgetedSVM toolbox	124
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ llsvm.cpp	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ llsvm.h Defines classes and functions used for training and testing of LLSVM algorithm	138
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ mm_algs.cpp	??
C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/ mm_algs.h Defines classes and functions used for training and testing of large-scale multi-hyperplane algo- rithms (AMM batch, AMM online, and Pegasos)	142

Chapter 5

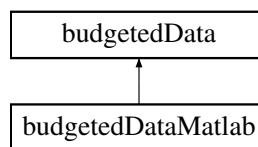
Class Documentation

5.1 budgetedData Class Reference

Class which handles manipulation of large data sets that cannot be fully loaded to memory (using a data structure similar to Matlab's sparse matrix structure).

```
#include <budgetedSVM.h>
```

Inheritance diagram for budgetedData:



Public Member Functions

- unsigned int [getDataDimensionality](#) (void)
Get the dimensionality of the data set.
- double [getSparsity](#) (void)
Get the sparsity of the data set (i.e., percentage of non-zero features). It is a number between 0 and 100, showing the sparsity in percentage points.
- unsigned int [getNumLoadedDataPointsSoFar](#) (void)
Get total number of data points loaded since the beginning of the epoch.
- [budgetedData](#) (bool [keepAssignments](#)=false, vector< int > [*yLabels](#)=NULL)
Vanilla constructor, just initializes the variables.
- [budgetedData](#) (const char fileName[], int dimension, unsigned int chunkSize, bool [keepAssignments](#)=false, vector< int > [*yLabels](#)=NULL)
Constructor that takes the data from LIBSVM-style .txt file.
- virtual [~budgetedData](#) (void)
Destructor, cleans up the memory.
- void [saveAssignment](#) (unsigned int *assigns)
Saves the current assignments, used by AMM batch.
- void [readChunkAssignments](#) (bool endOfFile)
Reads assignments for the current chunk, used by AMM batch.

- void `flushData` (void)
Clears all data taken up by the current chunk.
- virtual bool `readChunk` (unsigned int size, bool assign=false)
Reads the next data chunk.
- float `getElementOfVector` (unsigned int vector, unsigned int element)
Returns an element of a vector stored in `budgetedData` structure.
- long double `getVectorSqrL2Norm` (unsigned int vector, `parameters` *param)
Returns a squared L2-norm of a vector stored in `budgetedData` structure.
- double `distanceBetweenTwoPoints` (unsigned int index1, unsigned int index2)
Computes Euclidean distance between two data points from the input data.

Public Attributes

- unsigned long `loadTime`
Measures the time spent to load the data.
- vector< float > `an`
Vector of non-zero features of data points of the current data chunk. Where the data points start and end in this vector is specified by `ai` vector.
- vector< unsigned int > `aj`
Vector of indices of non-zero features of data points of the current data chunk. Where the data points start and end in this vector is specified by `ai` vector.
- vector< unsigned int > `ai`
Vector that tells us where the data point starts in vectors `an` and `aj`, always of length `N`.
- unsigned char * `al`
Array of labels of the current data chunk, always of length `N`.
- vector< int > `yLabels`
Vector of possible labels, either found during loading or initialized during testing phase by the learned model.
- unsigned int `N`
Number of data points loaded.
- unsigned int * `assignments`
Assignments for the current data chunk, used for AMM batch algorithm.

Protected Attributes

- FILE * `ifile`
Pointer to a FILE object that identifies input data stream.
- FILE * `fAssignFile`
Pointer to a FILE object that identifies data stream of current assignments, used for AMM batch algorithm.
- const char * `ifileName`
Filename of LIBSVM-style .txt file with input data.
- const char * `ifileNameAssign`
Filename of .txt file that keeps current assignments of weights to input data points, used for AMM batch algorithm.
- unsigned int `dimensionHighestSeen`
Highest dimension seen during loading of the data, or is equal to the user-specified dimensionality of the data. It does not include bias term and holds only the true, original dimensionality of the input data, even if bias term parameter is set to a non-zero value.
- unsigned int `numNonZeroFeatures`
Number of non-zero features of the currently loaded chunk, found during loading of the data. Used to compute the sparsity of the data.
- unsigned int `loadedDataPointsSoFar`

- *Total number of data points loaded so far.*
- bool `fileOpened`
Indicates that the input data .txt file is open.
- bool `fileAssignOpened`
Indicates that the .txt file with current assignments is open, used for AMM batch algorithm.
- bool `dataPartiallyLoaded`
Indicates that the data is only partially loaded to memory. It can also be fully loaded, e.g., when using data already loaded by some other application, Matlab for instance.
- bool `keepAssignments`
Indicates that assignments should be kept, true only for AMM batch algorithm.
- bool `isTrainingSet`
Set to true if loading the training data set, set false when loading testing data set. Affects the population of `yLabels` array that holds the possible labels in the data set: during training phase every previously unseen label is added to the array of possible labels, while during testing phase a warning message is printed when previously unseen label is found.

5.1.1 Detailed Description

Class which handles manipulation of large data sets that cannot be fully loaded to memory (using a data structure similar to Matlab's sparse matrix structure).

In order to handle large data sets, we do not load the entire data into memory, instead load it in smaller chunks. The loaded chunk is stored in a structure similar to Matlab's sparse matrix structure. Namely, only non-zero features and corresponding feature values of data points are stored in one budget vector for fast access, with additional vector that hold pointers to feature vector telling us where the information for each individual data point starts.

Definition at line 334 of file `budgetedSVM.h`.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 `budgetedData()` [1/2]

```
budgetedData::budgetedData (
    bool keepAssignments = false,
    vector< int > * yLabels = NULL )
```

Vanilla constructor, just initializes the variables.

Parameters

in	<code>keepAssignments</code>	True for AMM batch, otherwise false. File 'temp_assigns.txt' will be created and deleted to keep the assignments.
in	<code>yLabels</code>	Possible labels in the classification problem, for training data is NULL since they are inferred from data.

Definition at line 204 of file `budgetedSVM.cpp`.

```
205 {
206     this->ifileName = NULL;
```

```

207     this->ifileNameAssign = NULL;
208     this->dimensionHighestSeen = 0;
209     this->ifile = NULL;
210     this->assignments = NULL;
211     this->a1 = NULL;
212     this->keepAssignments = keepAssignments;
213     this->loadTime = 0;
214     this->N = 0;
215     this->dataPartiallyLoaded = false;
216     this->loadedDataPointsSoFar = 0;
217     this->numNonZeroFeatures = 0;
218     this->isTrainingSet = true;
219
220     // if labels provided use them, this happens in the case of testing data
221     if (yLabels)
222     {
223         for (unsigned int i = 0; i < (*yLabels).size(); i++)
224         {
225             this->yLabels.push_back((*yLabels)[i]);
226         }
227
228         this->isTrainingSet = false;
229     }
230 }

```

5.1.2.2 budgetedData() [2/2]

```

budgetedData::budgetedData (
    const char fileName[],
    int dimension,
    unsigned int chunkSize,
    bool keepAssignments = false,
    vector< int > * yLabels = NULL )

```

Constructor that takes the data from LIBSVM-style .txt file.

Parameters

in	<i>fileName</i>	Path to the input .txt file.
in	<i>dimension</i>	Dimensionality of the classification problem.
in	<i>chunkSize</i>	Size of the input data chunk that is loaded.
in	<i>keepAssignments</i>	True for AMM batch, otherwise false. File 'temp_assigns.txt' will be created and deleted to keep the assignments.
in	<i>yLabels</i>	Possible labels in the classification problem, for training data is NULL since inferred from data.

Definition at line 240 of file budgetedSVM.cpp.

```

241 {
242     this->isTrainingSet = true;
243     this->ifileName = strdup(fileName);
244     if (dimension < 1)
245         // if the input data dimensionality is incorrectly set, then we will infer the data
246         // dimensionality during data loading
247         this->dimensionHighestSeen = 0;
248     else
249         this->dimensionHighestSeen = dimension;
250
251     this->a1 = new (nothrow) unsigned char[chunkSize];
252     if (this->a1 == NULL)
253     {
254         svmPrintErrorString("Memory allocation error (budgetedData Constructor)!");
255     }
256
257     // keepAssignments is used for AMM_batch, where we hold the epoch assignments of data points to
258     // hyperplanes
259     this->keepAssignments = keepAssignments;
260     if (keepAssignments)

```

```

259     {
260         this->ifileNameAssign = strdup("temp_assigns.txt"); // here we set name of the file in
which the temporary assignments are kept; it will be removed after the training is completed
261         this->assignments = new (nothrow) unsigned int[chunkSize];
262     }
263     else
264         this->assignments = NULL;
265
266     // if labels provided use them, this happens in the case of testing data
267     if (yLabels)
268     {
269         for (unsigned int i = 0; i < (*yLabels).size(); i++)
270         {
271             this->yLabels.push_back((*yLabels)[i]);
272         }
273         this->isTrainingSet = false;
274     }
275
276     this->fileOpened = false;
277     this->fileAssignOpened = false;
278     this->loadTime = 0;
279     this->N = 0;
280     this->dataPartiallyLoaded = true;
281     this->loadedDataPointsSoFar = 0;
282     this->numNonZeroFeatures = 0;
283 }

```

5.1.3 Member Function Documentation

5.1.3.1 distanceBetweenTwoPoints()

```
double budgetedData::distanceBetweenTwoPoints (
    unsigned int index1,
    unsigned int index2 )
```

Computes Euclidean distance between two data points from the input data.

Parameters

in	<i>index1</i>	Index of the first data point.
in	<i>index2</i>	Index of the second data point.

Returns

Euclidean distance between the two points.

Definition at line 580 of file budgetedSVM.cpp.

```

581 {
582     // if distance to itself, return 0.0
583     if (index1 == index2)
584         return 0.0;
585
586     long icurrent1 = ai[index1];
587     long iend1 = (index1 == ai.size() - 1) ? aj.size() : ai[index1 + 1];
588     long icurrent2 = ai[index2];
589     long iend2 = (index2 == ai.size() - 1) ? aj.size() : ai[index2 + 1];
590     double dotxx = 0.0, dotyy = 0.0, dotxy = 0.0;
591
592     double currFeat1, currFeat2;
593     while (1)
594     {
595         // traverse the vectors non-zero feature by non-zero feature
596         if (icurrent1 < iend1)
597             currFeat1 = (double) aj[icurrent1];

```

```

598     else
599         currFeat1 = INF;
600     if (icurrent2 < iend2)
601         currFeat2 = (double) aj[icurrent2];
602     else
603         currFeat2 = INF;
604
605     if (currFeat1 == currFeat2)
606     {
607         dotxy += (an[icurrent1] * an[icurrent2]);
608         dotxx += (an[icurrent1] * an[icurrent1]);
609         dotyy += (an[icurrent2] * an[icurrent2]);
610
611         icurrent1++;
612         icurrent2++;
613     }
614     else
615     {
616         if (currFeat1 < currFeat2)
617         {
618             dotxx += (an[icurrent1] * an[icurrent1]);
619             icurrent1++;
620         }
621         else
622         {
623             dotyy += (an[icurrent2] * an[icurrent2]);
624             icurrent2++;
625         }
626     }
627
628     if ((icurrent1 >= iend1) && (icurrent2 >= iend2))
629         break;
630 }
631 return dotxx + dotyy - 2.0 * dotxy;
632 }

```

5.1.3.2 getDataDimensionality()

```

unsigned int budgetedData::getDataDimensionality (
    void ) [inline]

```

Get the dimensionality of the data set.

Returns

Returns the dimensionality of the data set.

Definition at line 425 of file budgetedSVM.h.

```

426     {
427         return dimensionHighestSeen;
428     };

```

5.1.3.3 getElementOfVector()

```

float budgetedData::getElementOfVector (
    unsigned int vector,
    unsigned int element )

```

Returns an element of a vector stored in [budgetedData](#) structure.

Parameters

in	<i>vector</i>	Index of the vector (C-style indexing used, starting from 0; note that LibSVM format indices start from 1).
in	<i>element</i>	Index of the element of the vector (C-style indexing used, starting from 0; note that LibSVM format indices start from 1).

Returns

Element of the vector specified as an input.

In the case that we need to read an element of a vector from currently loaded data chunk, we can use this function to access these vector elements.

Definition at line 509 of file budgetedSVM.cpp.

```

510 {
511     unsigned int maxPointIndex, pointIndexPointer;
512
513     // check if vector index too big
514     if (vector >= this->N)
515     {
516         svmPrintString("Warning: Vector index in getElementOfVector() function out of bounds, returning
default value of 0.\n");
517         return 0.0;
518     }
519     // check if element index too big
520     if (element >= this->dimensionHighestSeen)
521     {
522         svmPrintString("Warning: Element index in getElementOfVector() function out of bounds, returning
default value of 0.\n");
523         return 0.0;
524     }
525
526     pointIndexPointer = this->ai[vector];
527     maxPointIndex = ((unsigned int)(vector + 1) == this->N) ? (unsigned int) (this->aj.size()) :
this->ai[vector + 1];
528
529     for (unsigned int i = pointIndexPointer; i < maxPointIndex; i++)
530     {
531         // if we found the element return its value
532         if (this->aj[i] == element + 1)
533             return this->an[i];
534
535         // if we went over the index of the wanted element, then the element is equal to 0
536         if (this->aj[i] > element + 1)
537             return 0.0;
538     }
539     // if the wanted element is indexed higher than all non-zero elements, then it is equal to 0
540     return 0.0;
541 }

```

5.1.3.4 getNumLoadedDataPointsSoFar()

```

unsigned int budgetedData::getNumLoadedDataPointsSoFar (
    void ) [inline]

```

Get total number of data points loaded since the beginning of the epoch.

Returns

Number of data points loaded since the beginning of the epoch.

Definition at line 443 of file budgetedSVM.h.

```

444     {
445         return loadedDataPointsSoFar;
446     };

```

5.1.3.5 getSparsity()

```
double budgetedData::getSparsity (
    void ) [inline]
```

Get the sparsity of the data set (i.e., percentage of non-zero features). It is a number between 0 and 100, showing the sparsity in percentage points.

Returns

Returns the sparsity of the data set in percentage points.

Definition at line 434 of file budgetedSVM.h.

```
435     {
436         return (100.0 * (double) numNonZeroFeatures / ((double) loadedDataPointsSoFar * (double)
            dimensionHighestSeen));
437     };
```

5.1.3.6 getVectorSqrL2Norm()

```
long double budgetedData::getVectorSqrL2Norm (
    unsigned int vector,
    parameters * param )
```

Returns a squared L2-norm of a vector stored in [budgetedData](#) structure.

Parameters

in	<i>vector</i>	Index of the vector (C-style indexing used, starting from 0; note that LibSVM format indices start from 1).
in	<i>param</i>	The parameters of the algorithm.

Returns

Squared L2-norm of a vector.

This function returns squared L2-norm of a vector stored in the [budgetedData](#) structure. In particular, it is used to speed up the computation of Gaussian kernel.

Definition at line 551 of file budgetedSVM.cpp.

```
552 {
553     unsigned int maxPointIndex, pointIndexPointer;
554     long double result = 0.0;
555
556     // check if vector index too big
557     if (vector >= this->N)
558     {
559         svmPrintString("Warning: Vector index in getElementOfVector() function out of bounds, returning
            default value of 0.\n");
560         return 0.0;
561     }
562
563     pointIndexPointer = this->ai[vector];
564     maxPointIndex = ((unsigned int)(vector + 1) == this->N) ? (unsigned int)(this->aj.size()) :
            this->ai[vector + 1];
565
566     for (unsigned int i = pointIndexPointer; i < maxPointIndex; i++)
```



```

567         result += (this->an[i] * this->an[i]);
568         if (param->BIAS_TERM != 0.0)
569             result += (param->BIAS_TERM * param->BIAS_TERM);
570
571         return result;
572 }

```

5.1.3.7 readChunk()

```

bool budgetedData::readChunk (
    unsigned int size,
    bool assign = false ) [virtual]

```

Reads the next data chunk.

Parameters

in	<i>size</i>	Size of the chunk (i.e., number of data points) to be loaded.
in	<i>assign</i>	True if assignments should be saved, false otherwise.

Returns

True if just read the last data chunk, false otherwise.

In order to handle large data sets, we do not load the entire data into memory, instead load it in smaller chunks. Once we have finished processing a loaded data chunk, we load a new one using this function. The return value tells us if there are more chunks left; while there is still data to be loaded the function returns false, if we are done with the data set the function returns true. In the case of the AMM_batch algorithm, we also need to store current assignments of data points to weights, if the input "assign" is true then the function also initializes a .txt file for purpose of storing these assignments when the first chunk is loaded.

Reimplemented in [budgetedDataMatlab](#).

Definition at line 382 of file budgetedSVM.cpp.

```

383 {
384     string text;
385
386     char line[262143]; // maximum length of the line to be read is set to 262143
387     char str[256];
388     int pos, label;
389     unsigned int counter = 0, dimSeen, pointIndex = 0;
390     unsigned long start = clock();
391     bool labelFound, warningWritten = false;
392
393     // if not loaded from .txt file just exit
394     if (!dataPartiallyLoaded)
395         return false;
396
397     flushData();
398     if (!fileOpened)
399     {
400         this->ifile = fopen(ifileName, "rt");
401         this->fileOpened = true;
402         this->loadedDataPointsSoFar = 0;
403         this->numNonZeroFeatures = 0;
404
405         // if the very beginning, just create the assignment file if necessary
406         if ((!assign) && (keepAssignments))
407         {
408             fAssignFile = fopen(ifileNameAssign, "wt");
409             fclose(fAssignFile);
410         }
411     }
412 }

```

```

413 // load chunk
414 while (fgets(line, 262143, ifile))
415 {
416     N++;
417     loadedDataPointsSoFar++;
418
419     stringstream ss;
420     ss << line;
421
422     // get label
423     if (ss >> text)
424     {
425         label = atoi(text.c_str());
426         ai.push_back(pointIndex);
427
428         // get yLabels, if label not seen before add it into the label array
429         labelFound = false;
430         for (unsigned int i = 0; i < yLabels.size(); i++)
431         {
432             if (yLabels[i] == label)
433             {
434                 al[counter++] = (char) i;
435                 labelFound = true;
436                 break;
437             }
438         }
439
440         if (!labelFound)
441         {
442             if (isTrainingSet)
443             {
444                 yLabels.push_back(label);
445                 al[counter++] = (char) (yLabels.size() - 1);
446             }
447             else
448             {
449                 // so unseen label detected during testing phase, issue a warning
450                 if (!warningWritten)
451                 {
452                     sprintf(str, "Warning: Testing label '%d' detected during loading that was not
seen in training.\n", label);
453                     svmPrintString(str);
454                     warningWritten = true;
455                 }
456
457                 // give an example a label index that can never be predicted
458                 al[counter++] = (char) yLabels.size();
459             }
460         }
461     }
462
463     // get feature values
464     while (ss >> text)
465     {
466         if ((pos = (int) text.find(":"))
467         {
468             dimSeen = atoi(text.substr(0, pos).c_str());
469             aj.push_back(dimSeen);
470             an.push_back((float) atof(text.substr(pos + 1, text.length()).c_str()));
471             pointIndex++;
472             numNonZeroFeatures++;
473
474             // if more features found than specified, print error message
475             /*if (dimensionHighestSeen < dimSeen)
476             {
477                 sprintf(line, "Found more features than specified with '-D' option (specified: %d,
found %d)!\nPlease check your settings.\n", dimension, dimSeen);
478                 svmPrintErrorString(line);
479             }*/
480
481             if (dimensionHighestSeen < dimSeen)
482                 dimensionHighestSeen = dimSeen;
483         }
484     }
485
486     // check the size of chunk
487     if (N == size)
488     {
489         // still data left to load, keep working
490         loadTime += (clock() - start);
491         return true;
492     }
493 }
494
495 // got to the end of file, no more data left to load, exit nicely
496 fclose(ifile);
497 fileOpened = false;

```

```

498     loadTime += (clock() - start);
499
500     return false;
501 }

```

5.1.3.8 readChunkAssignments()

```

void budgetedData::readChunkAssignments (
    bool endOfFile )

```

Reads assignments for the current chunk, used by AMM batch.

Parameters

in	<i>endOfFile</i>	If the final chunk, close the assignment file.
----	------------------	--

During AMM batch training phase we need to keep track of the assignment of non-zero weights to data points. We store the assignments into a text file and load them together with the data chunk currently loaded, as it may be expensive to store all assignments in memory when working with large data sets.

Definition at line 334 of file budgetedSVM.cpp.

```

335 {
336     // if data is fully loaded from the beginning then just exit (e.g., can happen when BudgetedSVM is
    called from Matlab interface)
337     if (!dataPartiallyLoaded)
338         return;
339
340     int tempInt;
341     if (!fileAssignOpened)
342     {
343         fileAssignOpened = true;
344         fAssignFile = fopen(ifileNameAssign, "rt");
345     }
346
347     for (unsigned int i = 0; i < N; i++)
348     {
349         // get the assignments (as opposed to initial iteration and reassignment phase
350         // where we write the assignments, here we read them)
351         if (!fscanf(fAssignFile, "%d\n", &tempInt))
352         {
353             svmPrintErrorString("Error reading assignments from the text file!\n");
354         }
355         *(assignments + i) = (unsigned int) tempInt;
356     }
357
358     if (endOfFile)
359     {
360         fileAssignOpened = false;
361         fclose(fAssignFile);
362     }
363 };

```

5.1.3.9 saveAssignment()

```

void budgetedData::saveAssignment (
    unsigned int * assigns )

```

Saves the current assignments, used by AMM batch.

Parameters

in	<i>assigns</i>	Current assignments.
----	----------------	----------------------

Definition at line 308 of file budgetedSVM.cpp.

```

309 {
310     // no need for saving and loading to file, if data is fully (i.e., not partially) loaded, then
    everything is in the workspace (e.g., in the case of Matlab interface this can happen)
311     if (!dataPartiallyLoaded)
312     {
313         if (assignments == NULL)
314             assignments = new (nothrow) unsigned int[N];
315
316         for (unsigned int i = 0; i < N; i++)
317             *(assignments + i) = *(assigns + i);
318
319         return;
320     }
321
322     fAssignFile = fopen(ifileNameAssign, "at");
323
324     for (unsigned int i = 0; i < N; i++)
325         fprintf(fAssignFile, "%d\n", *(assigns + i));
326
327     fclose(fAssignFile);
328 };

```

5.1.4 Member Data Documentation

5.1.4.1 assignments

unsigned int * budgetedData::assignments

Assignments for the current data chunk, used for AMM batch algorithm.

See also

[fAssignFile](#)

Definition at line 419 of file budgetedSVM.h.

5.1.4.2 dimensionHighestSeen

long budgetedData::dimensionHighestSeen [protected]

Highest dimension seen during loading of the data, or is equal to the user-specified dimensionality of the data. It does not include bias term and holds only the true, original dimensionality of the input data, even if bias term parameter is set to a non-zero value.

See also

[parameters::BIAS_TERM](#)

Definition at line 408 of file budgetedSVM.h.

5.1.4.3 fAssignFile

```
FILE * budgetedData::fAssignFile [protected]
```

Pointer to a FILE object that identifies data stream of current assignments, used for AMM batch algorithm.

During AMM batch training phase we need to keep track of which non-zero weight is assigned to which data point. We store the assignments into text file and load them together with the data chunk currently loaded, as it might be to expensive to store all assignments in memory. In order to keep track of this weight-example mapping, each weight vector also has a unique [budgetedVector::weightID](#), assigned to each vector upon creation.

See also

[parameters::CHUNK_SIZE](#)
[budgetedVector::weightID](#)

Definition at line 406 of file budgetedSVM.h.

5.1.4.4 ifileNameAssign

```
const char * budgetedData::ifileNameAssign [protected]
```

Filename of .txt file that keeps current assignments of weights to input data points, used for AMM batch algorithm.

During AMM batch training phase we need to keep track of which non-zero weight is assigned to which data point. We store the assignments into text file and load them together with the data chunk currently loaded, as it might be to expensive to store all assignments in memory.

See also

[parameters::CHUNK_SIZE](#)

Definition at line 407 of file budgetedSVM.h.

The documentation for this class was generated from the following files:

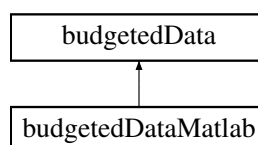
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/[budgetedSVM.h](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/[budgetedSVM.cpp](#)

5.2 budgetedDataMatlab Class Reference

Class which manipulates sparse array of vectors (similarly to Matlab sparse matrix structure), with added functionality to load data directly from Matlab.

```
#include <budgetedSVM_matlab.h>
```

Inheritance diagram for budgetedDataMatlab:



Public Member Functions

- bool [readChunk](#) (unsigned int size, bool assign=false)
Overrides virtual function from [budgetedData](#), simply returns false regardless of inputs as the data is fully loaded from Matlab.
- [budgetedDataMatlab](#) (const mxArray *labelVec, const mxArray *instanceMat, [parameters](#) *param, bool keepAssignments=false, vector< int > *yLabels=NULL)
Constructor, invokes [readDataFromMatlab](#) that loads Matlab data.
- [~budgetedDataMatlab](#) (void)
Destructor, cleans up the memory.

Protected Member Functions

- void [readDataFromMatlab](#) (const mxArray *labelVec, const mxArray *instanceMat, [parameters](#) *param)
Loads the data from Matlab.

Additional Inherited Members

5.2.1 Detailed Description

Class which manipulates sparse array of vectors (similarly to Matlab sparse matrix structure), with added functionality to load data directly from Matlab.

Class which manipulates sparse array of vectors (similarly to Matlab sparse matrix structure), with added functionality to load data directly from Matlab. Unlike [budgetedData](#), where we load the data in smaller chunks, in this class we assume that the entire data can be loaded into memory, as it is already loaded in Matlab.

Definition at line 28 of file `budgetedSVM_matlab.h`.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 [budgetedDataMatlab\(\)](#)

```
budgetedDataMatlab::budgetedDataMatlab (
    const mxArray * labelVec,
    const mxArray * instanceMat,
    parameters * param,
    bool keepAssignments = false,
    vector< int > * yLabels = NULL ) [inline]
```

Constructor, invokes [readDataFromMatlab](#) that loads Matlab data.

Parameters

in	<i>labelVec</i>	Vector of labels.
in	<i>instanceMat</i>	Matrix of data points, each row is a single data point.
in	<i>param</i>	The parameters of the algorithm.
in	<i>keepAssignments</i>	True for AMM batch, otherwise false. Unlike in budgetedData case, no file is created to store the assignments as it is assumed that the memory to hold the assignments can be allocated in whole.
in	<i>yLabels</i>	Possible labels in the classification problem, for training data is NULL since inferred from data.

Definition at line 59 of file budgetedSVM_matlab.h.

```

59                                     : budgetedData(keepAssignments, yLabels)
60     {
61         readDataFromMatlab(labelVec, instanceMat, param);
62     };

```

5.2.3 Member Function Documentation

5.2.3.1 readChunk()

```

bool budgetedDataMatlab::readChunk (
    unsigned int size,
    bool assign = false ) [inline], [virtual]

```

Overrides virtual function from [budgetedData](#), simply returns false regardless of inputs as the data is fully loaded from Matlab.

Parameters

in	<i>size</i>	Size of the chunk to be loaded.
in	<i>assign</i>	True if assignment should be saved, false otherwise.

Returns

False regardless of inputs, since the data is fully loaded from Matlab.

Reimplemented from [budgetedData](#).

Definition at line 46 of file budgetedSVM_matlab.h.

```

47     {
48         return false;
49     };

```

5.2.3.2 readDataFromMatlab()

```

void budgetedDataMatlab::readDataFromMatlab (
    const mxArray * labelVec,
    const mxArray * instanceMat,
    parameters * param ) [protected]

```

Loads the data from Matlab.

Parameters

in	<i>labelVec</i>	Vector of labels.
in	<i>instanceMat</i>	Matrix of data points, each row is a single data point.
in	<i>param</i>	The parameters of the algorithm.

Definition at line 82 of file budgetedSVM_matlab.cpp.

```

83 {
84     long start = clock();
85     unsigned int i, j, k, labelVectorRowNum;
86     long unsigned int low, high;
87     mwIndex *ir, *jc;
88     double *samples, *labels;
89     bool labelFound;
90     mxArray *instanceMatCol; // transposed instance sparse matrix
91     bool warningWritten = false;
92     char str[256];
93
94     // otherwise load the data, given below
95     // transpose instance matrix
96     {
97         mxArray *prhs[1], *plhs[1];
98         prhs[0] = mxDuplicateArray(instanceMat);
99         if (mexCallMATLAB(1, plhs, 1, prhs, "transpose"))
100             mexErrMsgTxt("Error: Cannot transpose training instance matrix.\n");
101
102         instanceMatCol = plhs[0];
103         mxDestroyArray(prhs[0]);
104     }
105
106     // each column is one instance
107     labels = mxGetPr(labelVec);
108     samples = mxGetPr(instanceMatCol);
109
110     // get number of instances
111     labelVectorRowNum = (int)mxGetM(labelVec);
112     if (labelVectorRowNum != (int)mxGetN(instanceMatCol))
113         mexErrMsgTxt("Length of label vector does not match number of instances.\n");
114
115     // set the dimension and the number of data points
116     this->N = labelVectorRowNum;
117     if ((*param).DIMENSION == 0)
118     {
119         // it is 0 when loading training data set
120         this->dimensionHighestSeen = (*param).DIMENSION = (int)mxGetM(instanceMatCol);
121         if ((*param).BIAS_TERM != 0.0)
122             (*param).DIMENSION++;
123
124         // set KERNEL_GAMMA_PARAM here if needed, done during loading of training set
125         if ((*param).KERNEL_GAMMA_PARAM == 0.0)
126             (*param).KERNEL_GAMMA_PARAM = 1.0 / (double) (*param).DIMENSION;
127     }
128     else
129     {
130         // it is non-zero only when loading testing data set, no need to set GAMMA parameter as it is
131         // read from the model structure from Matlab
132         this->dimensionHighestSeen = (*param).DIMENSION;
133
134         // if bias term is non-zero, then the actual dimensionality of data is one less than DIMENSION
135         if ((*param).BIAS_TERM != 0.0)
136             this->dimensionHighestSeen--;
137     }
138
139     // allocate memory for labels
140     this->al = new (nothrow) unsigned char[this->N];
141     if (this->al == NULL)
142         mexErrMsgTxt("Memory allocation error (readDataFromMatlab function)! Restart MATLAB and try again.");
143
144     if (mxIsSparse(instanceMat))
145     {
146         ir = mxGetIr(instanceMatCol);
147         jc = mxGetJc(instanceMatCol);
148
149         j = 0;
150         for (i = 0; i < labelVectorRowNum; i++)
151         {
152             // where the instance starts
153             ai.push_back(j);
154
155             // get yLabels, if label not seen before add it in the label array
156             labelFound = false;
157             for (k = 0; k < (int) yLabels.size(); k++)
158             {
159                 if (yLabels[k] == (int)labels[i])
160                 {
161                     al[i] = k;
162                     labelFound = true;
163                     break;
164                 }
165             }
166             if (!labelFound)

```



```

167         if (isTrainingSet)
168         {
169             yLabels.push_back((int)labels[i]);
170             al[i] = (unsigned char) (yLabels.size() - 1);
171         }
172         else
173         {
174             // so unseen label detected during testing phase, issue a warning
175             if (!warningWritten)
176             {
177                 sprintf(str, "Warning: Testing label '%d' detected that was not seen during
training.\n", (int)labels[i]);
178                 mexPrintf(str);
179                 mexEvalString("drawnow;");
180
181                 warningWritten = true;
182             }
183
184             // give an example a label index that can never be predicted
185             al[i] = (unsigned char) yLabels.size();
186         }
187     }
188
189     // get features
190     low = (int) jc[i], high = (int) jc[i + 1];
191     for (k = low; k < high; k++)
192     {
193         // we save the actual feature no. in aj, and the value in an
194         aj.push_back((int) ir[k] + 1);
195         an.push_back((float) samples[k]);
196         j++;
197     }
198 }
199 }
200 else
201 {
202     j = 0;
203     low = 0;
204     for (i = 0; i < labelVectorRowNum; i++)
205     {
206         // where the instance starts
207         al.push_back(j);
208
209         // get yLabels, if label not seen before add it in the label array
210         labelFound = false;
211         for (k = 0; k < (int) yLabels.size(); k++)
212         {
213             if (yLabels[k] == (int) labels[i])
214             {
215                 al[i] = k;
216                 labelFound = true;
217                 break;
218             }
219         }
220         if (!labelFound)
221         {
222             if (isTrainingSet)
223             {
224                 yLabels.push_back((int)labels[i]);
225                 al[i] = (unsigned char) (yLabels.size() - 1);
226             }
227             else
228             {
229                 // so unseen label detected during testing phase, issue a warning
230                 if (!warningWritten)
231                 {
232                     sprintf(str, "Warning: Testing label '%d' detected that was not seen during
training.\n", (int) labels[i]);
233                     mexPrintf(str);
234                     mexEvalString("drawnow;");
235
236                     warningWritten = true;
237                 }
238
239                 // give an example a label index that can never be predicted
240                 al[i] = (unsigned char) yLabels.size();
241             }
242         }
243
244         // get features
245         for (k = 0; k < (int)mxGetM(instanceMatCol); k++)
246         {
247             if (samples[low] != 0.0)
248             {
249                 // we save the actual feature no. in aj, and the value in an
250                 aj.push_back(k + 1);
251                 an.push_back((float) samples[low]);

```

```

252             j++;
253         }
254         low++;
255     }
256 }
257 }
258
259 // if very beginning, just allocate memory for assignments
260 if (keepAssignments)
261     this->assignments = new (nothrow) unsigned int[this->N];
262
263 loadTime += (clock() - start);
264 };

```

The documentation for this class was generated from the following files:

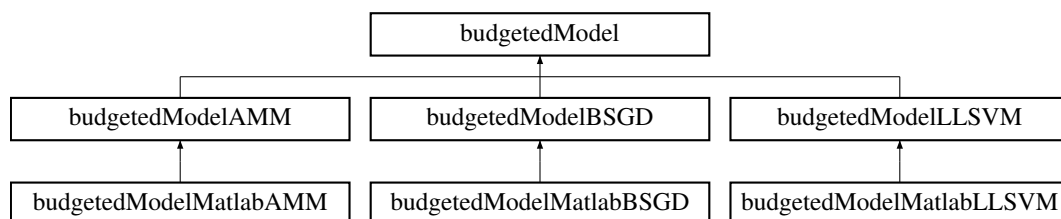
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.h](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.cpp](#)

5.3 budgetedModel Class Reference

Interface which defines methods to load model from and save model to text file.

```
#include <budgetedSVM.h>
```

Inheritance diagram for budgetedModel:



Public Member Functions

- virtual void [extendDimensionalityOfModel](#) (unsigned int newDim, [parameters](#) *param)
Extends the dimensionality of each support vector and hyperplane in the model.
- virtual [~budgetedModel](#) (void)
Destructor, cleans up the memory.
- virtual bool [saveToTextFile](#) (const char *filename, vector< int > *yLabels, [parameters](#) *param)=0
Saves the trained model to .txt file.
- virtual bool [loadFromTextFile](#) (const char *filename, vector< int > *yLabels, [parameters](#) *param)=0
Loads the trained model from .txt file.

Static Public Member Functions

- static int [getAlgorithm](#) (const char *filename)
Get algorithm code from the trained model stored in .txt file, according to enumeration explained at the top of this page.

5.3.1 Detailed Description

Interface which defines methods to load model from and save model to text file.

In order to ensure that all algorithms have the same interface when it comes to storing/loading of the trained model, this interface is to be implemented by each separate algorithm model.

Definition at line 875 of file budgetedSVM.h.

5.3.2 Member Function Documentation

5.3.2.1 extendDimensionalityOfModel()

```
void budgetedModel::extendDimensionalityOfModel (
    unsigned int newDim,
    parameters * param ) [inline], [virtual]
```

Extends the dimensionality of each support vector and hyperplane in the model.

Parameters

in	<i>newDim</i>	Filename of the .txt file where the model is saved.
in	<i>param</i>	Parameters of the algorithm.

Extends the dimensionality of each support vector and hyperplane in the model. Called after new data chunk has been loaded, could be needed when user set the dimensionality of the data incorrectly, and we infer this important parameter during loading of the data.

Reimplemented in [budgetedModelAMM](#), [budgetedModelBSGD](#), and [budgetedModelLLSVM](#).

Definition at line 892 of file budgetedSVM.h.

```
892 {};
```

5.3.2.2 getAlgorithm()

```
static int budgetedModel::getAlgorithm (
    const char * filename ) [static]
```

Get algorithm code from the trained model stored in .txt file, according to enumeration explained at the top of this page.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
----	-----------------	---

Returns

-1 if error, otherwise returns algorithm code from the model file.

Definition at line 182 of file budgetedSVM.cpp.

```

183 {
184     FILE *fModel = NULL;
185     int temp;
186     fModel = fopen(filename, "rt");
187     if (!fModel)
188         return -1;
189
190     if (!fscanf(fModel, "ALGORITHM: %d\n", &temp))
191     {
192         svmPrintErrorString("Error reading algorithm type from the model file!\n");
193     }
194     return temp;
195 }
```

5.3.2.3 loadFromTextFile()

```

bool budgetedModel::loadFromTextFile (
    const char * filename,
    vector< int > * yLabels,
    parameters * param ) [pure virtual]
```

Loads the trained model from .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
out	<i>yLabels</i>	Vector of possible labels.
out	<i>param</i>	Parameters of the algorithm.

Returns

False if error encountered, otherwise true.

The text file has the following rows: [ALGORITHM, DIMENSION, NUMBER_OF_CLASSES, LABELS, NUMBER_OF_WEIGHTS, BIAS_TERM, KERNEL_WIDTH, MODEL]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

- For AMM batch, AMM online, Pegasos: The model is stored so that each row of the text file corresponds to one weight. The first element of each weight is the class of the weight, followed by the degradation of the weight. The rest of the row corresponds to non-zero elements of the weight, given as *feature_index:feature_value*, in a standard LIBSVM format.
- For BSGD: The model is stored so that each row corresponds to one support vector (or weight). The first elements of each weight correspond to alpha parameters for each class, given in order specified by LABELS row. However, since alpha can be equal to 0, we use LIBSVM format to store alphas, as *-class_index: class-specific_alpha*, where we added '-' (minus sign) in front of the class index to differentiate between class indices and feature indices that follow. After the alphas, in the same row the elements of the weights (or support vectors) for each feature are given in LIBSVM format.
- For LLSVM: The model is stored so that each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space of the data set in LIBSVM format.

Implemented in [budgetedModelAMM](#), [budgetedModelBSGD](#), and [budgetedModelLLSVM](#).

5.3.2.4 saveToTextFile()

```
bool budgetedModel::saveToTextFile (
    const char * filename,
    vector< int > * yLabels,
    parameters * param ) [pure virtual]
```

Saves the trained model to .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	Parameters of the algorithm.

Returns

False if error encountered, otherwise true.

The text file has the following rows: [*ALGORITHM*, *DIMENSION*, *NUMBER_OF_CLASSES*, *LABELS*, *NUMBER_OF_WEIGHTS*, *BIAS_TERM*, *KERNEL_WIDTH*, *MODEL*]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

- For AMM batch, AMM online, Pegasos: The model is stored so that each row of the text file corresponds to one weight. The first element of each weight is the class of the weight, followed by the degradation of the weight. The rest of the row corresponds to non-zero elements of the weight, given as *feature_index:feature_value*, in a standard LIBSVM format.
- For BSGD: The model is stored so that each row corresponds to one support vector (or weight). The first elements of each weight correspond to alpha parameters for each class, given in order by *LABELS* row. However, since alpha can be equal to 0, we use LIBSVM format to store alphas, as *-class_index:class-specific_alpha*, where we added '-' (minus sign) in front of the class index to differentiate between class indices and feature indices that follow. After the alphas, in the same row the elements of the weights (or support vectors) for each feature are given in LIBSVM format.
- For LLSVM: The model is stored so that each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space of the data set in LIBSVM format.

Implemented in [budgetedModelAMM](#), [budgetedModelBSGD](#), and [budgetedModelLLSVM](#).

Definition at line 899 of file `budgetedSVM.h`.

The documentation for this class was generated from the following files:

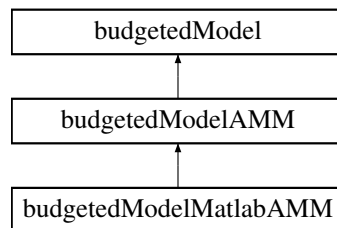
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/[budgetedSVM.h](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/[budgetedSVM.cpp](#)

5.4 budgetedModelAMM Class Reference

Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to text file.

```
#include <mm_algs.h>
```

Inheritance diagram for budgetedModelAMM:



Public Member Functions

- [budgetedModelAMM](#) (void)
Constructor, initializes the MM model to zero weights.
- [~budgetedModelAMM](#) (void)
Destructor, cleans up memory taken by AMM.
- [vector< vectorOfBudgetVectors > * getModel](#) (void)
Used to obtain a pointer to a current AMM model.
- [void extendDimensionalityOfModel](#) (unsigned int newDim, [parameters](#) *param)
Extends the dimensionality of each linear hyperplane in the AMM model.
- [bool saveToTextFile](#) (const char *filename, [vector< int > *yLabels](#), [parameters](#) *param)
Saves the trained AMM model to .txt file.
- [bool loadFromTextFile](#) (const char *filename, [vector< int > *yLabels](#), [parameters](#) *param)
Loads the trained AMM model from .txt file.

Protected Attributes

- [vector< vectorOfBudgetVectors > * modelMM](#)
Holds AMM batch, AMM online, or PEGASOS models.

Additional Inherited Members

5.4.1 Detailed Description

Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to text file.

Definition at line 200 of file mm_algs.h.

5.4.2 Member Function Documentation

5.4.2.1 extendDimensionalityOfModel()

```
void budgetedModelAMM::extendDimensionalityOfModel (
    unsigned int newDim,
    parameters * param ) [inline], [virtual]
```

Extends the dimensionality of each linear hyperplane in the AMM model.

Extends the dimensionality of each linear hyperplane in the AMM model. Called after new data chunk has been loaded, could be needed when user set the dimensionality of the data incorrectly, and we infer this important parameter during loading of the data.

Reimplemented from [budgetedModel](#).

Definition at line 236 of file mm_algs.h.

```
237     {
238         // extend the dimensionality of each weight vector
239         for (unsigned int i = 0; i < (*modelMM).size(); i++)
240             for (unsigned int j = 0; j < (*modelMM)[i].size(); j++)
241                 (*modelMM)[i][j]->extendDimensionality(newDim, param);
242     };
```

5.4.2.2 getModel()

```
vector< vectorOfBudgetVectors > * budgetedModelAMM::getModel (
    void ) [inline]
```

Used to obtain a pointer to a current AMM model.

Returns

A pointer to a current AMM model.

Definition at line 226 of file mm_algs.h.

```
227     {
228         return modelMM;
229     };
```

5.4.2.3 loadFromTextFile()

```
bool budgetedModelAMM::loadFromTextFile (
    const char * filename,
    vector< int > * yLabels,
    parameters * param ) [virtual]
```

Loads the trained AMM model from .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
out	<i>yLabels</i>	Vector of possible labels.
out	<i>param</i>	The parameters of the algorithm.

Returns

Returns false if error encountered, otherwise true.

The text file has the following rows: [ALGORITHM, DIMENSION, NUMBER_OF_CLASSES, LABELS, NUMBER_OF_WEIGHTS, BIAS_TERM, KERNEL_WIDTH, MODEL]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

Each row of the text file corresponds to one weight. The first element of each weight is the class of the weight, followed by the degradation of the weight. The rest of the row corresponds to non-zero elements of the weight, given as feature_index:feature_value, in a standard LIBSVM format.

Implements [budgetedModel](#).

Definition at line 140 of file mm_algs.cpp.

```

141 {
142     unsigned int i, j, tempInt, numClasses;
143     float tempFloat;
144     string text;
145     char oneWord[1024];
146     int pos;
147     vector<unsigned int> numWeights;
148     FILE *fModel = NULL;
149     fModel = fopen(filename, "rt");
150     bool doneReadingBool;
151     long double sqrNorm;
152
153     if (!fModel)
154         return false;
155
156     // algorithm
157     fseek (fModel, (long) strlen("ALGORITHM: "), SEEK_CUR);
158     if (!fscanf(fModel, "%d\n", &((*param).ALGORITHM)))
159     {
160         svmPrintErrorString("Error reading algorithm type from the model file!\n");
161     }
162
163     // dimension
164     fseek (fModel, (long) strlen("DIMENSION: "), SEEK_CUR);
165     if (!fscanf(fModel, "%d\n", &((*param).DIMENSION)))
166     {
167         svmPrintErrorString("Error reading dimensions from the model file!\n");
168     }
169
170     // number of classes
171     fseek (fModel, (long) strlen("NUMBER_OF_CLASSES: "), SEEK_CUR);
172     if (!fscanf(fModel, "%d\n", &numClasses))
173     {
174         svmPrintErrorString("Error reading number of classes from the model file!\n");
175     }
176
177     // labels
178     fseek (fModel, (long) strlen("LABELS: "), SEEK_CUR);
179     for (i = 0; i < numClasses; i++)
180     {
181         if (!fscanf(fModel, "%d ", &tempInt))
182         {
183             svmPrintErrorString("Error reading labels from the model file!\n");
184         }
185         (*yLabels).push_back(tempInt);
186     }
187
188     // number of weights
189     fseek (fModel, (long) strlen("NUMBER_OF_WEIGHTS: "), SEEK_CUR);
190     for (i = 0; i < numClasses; i++)
191     {
192         if (!fscanf(fModel, "%d\n", &tempInt))
193         {

```



```

194         svmPrintErrorString("Error reading number of weights from the model file!\n");
195     }
196     numWeights.push_back(tempInt);
197 }
198
199 // bias parameter
200 fseek(fModel, (long) strlen("BIAS_TERM: "), SEEK_CUR);
201 if (!fscanf(fModel, "%f\n", &tempFloat))
202 {
203     svmPrintErrorString("Error reading bias term from the model file!\n");
204 }
205 (*param).BIAS_TERM = tempFloat;
206
207 // kernel width (GAMMA) parameter
208 fseek(fModel, (long) strlen("KERNEL_WIDTH: "), SEEK_CUR);
209 if (!fscanf(fModel, "%f\n", &tempFloat))
210 {
211     svmPrintErrorString("Error reading kernel width from the model file!\n");
212 }
213 (*param).KERNEL_GAMMA_PARAM = tempFloat;
214
215 // load the model
216 fseek(fModel, (long) strlen("MODEL:\n") + 1, SEEK_CUR);
217 for (i = 0; i < numClasses; i++) // for every class
218 {
219     // add for each class an empty weight matrix
220     vector<budgetedVectorAMM*> tempV;
221     (*modelMM).push_back(tempV);
222
223     for (j = 0; j < numWeights[i]; j++) // for every weight
224     {
225         budgetedVectorAMM *eNew = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
226         sqrNorm = 0.0L;
227
228         // get degradation and features
229
230         // skip label, no need to read it explicitly since we know the number of weights of each
231         // class, found in numWeights vector
232         fgetWord(fModel, oneWord);
233
234         // get degradation
235         doneReadingBool = fgetWord(fModel, oneWord);
236         eNew->setDegradation((long double) atof(oneWord));
237
238         // get features
239         while (!doneReadingBool)
240         {
241             doneReadingBool = fgetWord(fModel, oneWord);
242             if (strlen(oneWord) == 0)
243                 continue;
244
245             text = oneWord;
246             if ((pos = (int) text.find(":"))
247             {
248                 tempInt = atoi(text.substr(0, pos).c_str());
249                 tempFloat = (float) atof(text.substr(pos + 1, text.length()).c_str());
250                 (*eNew)[tempInt - 1] = tempFloat;
251                 sqrNorm += (long double) (tempFloat * tempFloat);
252             }
253         }
254         eNew->setSqrL2norm(sqrNorm);
255
256         (*modelMM)[i].push_back(eNew);
257         eNew = NULL;
258     }
259 }
260
261 fclose(fModel);
262 return true;
263 }

```

5.4.2.4 saveToTextFile()

```

bool budgetedModelAMM::saveToTextFile (
    const char * filename,
    vector< int > * yLabels,
    parameters * param ) [virtual]

```

Saves the trained AMM model to .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.

Returns

Returns false if error encountered, otherwise true.

The text file has the following rows: [ALGORITHM, DIMENSION, NUMBER_OF_CLASSES, LABELS, NUMBER_OF_WEIGHTS, BIAS_TERM, KERNEL_WIDTH, MODEL]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

Each row of the text file corresponds to one weight. The first element of each weight is the class of the weight, followed by the degradation of the weight. The rest of the row corresponds to non-zero elements of the weight, given as feature_index:feature_value, in a standard LIBSVM format.

Implements [budgetedModel](#).

Definition at line 72 of file mm_algs.cpp.

```

73 {
74     unsigned int i, j, k;
75     FILE *fModel = NULL;
76     fModel = fopen(filename, "wt");
77
78     if (!fModel)
79         return false;
80
81     // algorithm
82     fprintf(fModel, "ALGORITHM: %d\n", (*param).ALGORITHM);
83
84     // dimension
85     fprintf(fModel, "DIMENSION: %d\n", (*param).DIMENSION);
86
87     // number of classes
88     fprintf(fModel, "NUMBER_OF_CLASSES: %d\n", (int) (*yLabels).size());
89
90     // labels
91     fprintf(fModel, "LABELS:");
92     for (i = 0; i < (*yLabels).size(); i++)
93         fprintf(fModel, " %d", (*yLabels)[i]);
94     fprintf(fModel, "\n");
95
96     // number of weights
97     fprintf(fModel, "NUMBER_OF_WEIGHTS:");
98     for (i = 0; i < (*modelMM).size(); i++)
99         fprintf(fModel, " %d", (int) (*modelMM)[i].size());
100    fprintf(fModel, "\n");
101
102    // bias parameter
103    fprintf(fModel, "BIAS_TERM: %f\n", (*param).BIAS_TERM);
104
105    // kernel width (GAMMA) parameter
106    fprintf(fModel, "KERNEL_WIDTH: 0.0\n");
107
108    // save the model
109    fprintf(fModel, "MODEL:\n");
110    for (i = 0; i < yLabels->size(); i++) // for every class
111    {
112        for (j = 0; j < (*modelMM)[i].size(); j++) // for every weight
113        {
114            // weight label
115            fprintf(fModel, "%d ", (*yLabels)[i]);
116
117            // degradation
118            fprintf(fModel, "%2.10f", (double)((*modelMM)[i][j])>getDegradation());
119
120            for (k = 0; k < (*param).DIMENSION; k++) // for every feature
121            {
122                if (((*modelMM)[i][j])[k] != 0.0)
123                    fprintf(fModel, " %d:%2.10f", k + 1, ((*modelMM)[i][j])[k]);
124            }
125            fprintf(fModel, "\n");

```

```

126     }
127 }
128
129     fclose(fModel);
130     return true;
131 }

```

The documentation for this class was generated from the following files:

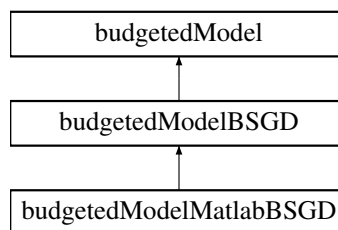
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.cpp

5.5 budgetedModelBSGD Class Reference

Class which holds the BSGD model (comprising the support vectors stored as [budgetedVectorBSGD](#)), and implements methods to load BSGD model from and save BSGD model to text file.

```
#include <bsgd.h>
```

Inheritance diagram for budgetedModelBSGD:



Public Member Functions

- void [extendDimensionalityOfModel](#) (unsigned int newDim, [parameters](#) *param)
Extends the dimensionality of each support vector in the BSGD model.
- [budgetedModelBSGD](#) (void)
Constructor, initializes the BSGD model to zero-vectors.
- [~budgetedModelBSGD](#) (void)
Destructor, cleans up memory taken by BSGD.
- bool [saveToTextFile](#) (const char *filename, vector< int > *yLabels, [parameters](#) *param)
Saves the trained BSGD model to .txt file.
- bool [loadFromTextFile](#) (const char *filename, vector< int > *yLabels, [parameters](#) *param)
Loads the trained BSGD model from .txt file.

Public Attributes

- vector< [budgetedVectorBSGD](#) * > * [modelBSGD](#)
Holds BSGD model.

Additional Inherited Members

5.5.1 Detailed Description

Class which holds the BSGD model (comprising the support vectors stored as [budgetedVectorBSGD](#)), and implements methods to load BSGD model from and save BSGD model to text file.

Definition at line 105 of file `bsgd.h`.

5.5.2 Member Function Documentation

5.5.2.1 `extendDimensionalityOfModel()`

```
void budgetedModelBSGD::extendDimensionalityOfModel (
    unsigned int newDim,
    parameters * param ) [inline], [virtual]
```

Extends the dimensionality of each support vector in the BSGD model.

Extends the dimensionality of each support vector in the BSGD model. Called after new data chunk has been loaded, could be needed when user set the dimensionality of the data incorrectly, and we infer this important parameter during loading of the data.

Reimplemented from [budgetedModel](#).

Definition at line 118 of file `bsgd.h`.

```
119     {
120         // extend the dimensionality of each weight vector
121         for (unsigned int i = 0; i < (*modelBSGD).size(); i++)
122             (*modelBSGD)[i]->extendDimensionality(newDim, param);
123     };
```

5.5.2.2 `loadFromTextFile()`

```
bool budgetedModelBSGD::loadFromTextFile (
    const char * filename,
    vector< int > * yLabels,
    parameters * param ) [virtual]
```

Loads the trained BSGD model from .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
out	<i>yLabels</i>	Vector of possible labels.
out	<i>param</i>	The parameters of the algorithm.

Returns

Returns false if error encountered, otherwise true.

The text file has the following rows: *[ALGORITHM, DIMENSION, NUMBER_OF_CLASSES, LABELS, NUMBER_OF_WEIGHTS, BIAS_TERM, KERNEL_WIDTH, MODEL]*. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

Each row corresponds to one support vector (or weight). The first elements of each weight correspond to alpha parameters for each class, given in order of "labels" member of the Matlab structure. However, since alpha can be equal to 0, we use LIBSVM format to store alphas, as -class_index:class-specific_alpha, where we added '-' (minus sign) in front of the class index to differentiate between class indices and feature indices that follow. After the alphas, in the same row the elements of the weights (or support vectors) for each feature are given in LIBSVM format.

Implements [budgetedModel](#).

Definition at line 147 of file bsgd.cpp.

```

148 {
149     unsigned int i, numClasses;
150     float tempFloat;
151     string text;
152     char oneWord[1024];
153     int pos, tempInt;
154     vector<unsigned int> numWeights;
155     FILE *fModel = NULL;
156     fModel = fopen(filename, "rt");
157     bool doneReadingBool;
158     long double sqrNorm;
159
160     if (!fModel)
161         return false;
162
163     // algorithm
164     fseek (fModel, strlen("ALGORITHM: "), SEEK_CUR);
165     if (!fscanf(fModel, "%d\n", &((*param).ALGORITHM)))
166     {
167         svmPrintErrorString("Error reading algorithm type from the model file!\n");
168     }
169
170     // dimension
171     fseek (fModel, strlen("DIMENSION: "), SEEK_CUR);
172     if (!fscanf(fModel, "%d\n", &((*param).DIMENSION)))
173     {
174         svmPrintErrorString("Error reading dimensions from the model file!\n");
175     }
176
177     // number of classes
178     fseek (fModel, strlen("NUMBER_OF_CLASSES: "), SEEK_CUR);
179     if (!fscanf(fModel, "%d\n", &numClasses))
180     {
181         svmPrintErrorString("Error reading number of classes from the model file!\n");
182     }
183
184     // labels
185     fseek (fModel, strlen("LABELS: "), SEEK_CUR);
186     for (i = 0; i < numClasses; i++)
187     {
188         if (!fscanf(fModel, "%d ", &tempInt))
189         {
190             svmPrintErrorString("Error reading labels from the model file!\n");
191         }
192         (*yLabels).push_back(tempInt);
193     }
194
195     // number of weights
196     fseek (fModel, strlen("NUMBER_OF_WEIGHTS: "), SEEK_CUR);
197     if (!fscanf(fModel, "%d\n", &tempInt))
198     {
199         svmPrintErrorString("Error reading number of weight from the model file!\n");
200     }
201     numWeights.push_back(tempInt);
202
203     // bias parameter
204     fseek (fModel, strlen("BIAS_TERM: "), SEEK_CUR);
205     if (!fscanf(fModel, "%f\n", &tempFloat))
206     {
207         svmPrintErrorString("Error reading bias term from the model file!\n");
208     }
209     (*param).BIAS_TERM = tempFloat;

```

```

210
211 // read kernel function
212 fseek (fModel, strlen("KERNEL_FUNCTION: "), SEEK_CUR);
213 if (!fscanf(fModel, "%d\n", &tempInt))
214 {
215     svmPrintErrorString("Error reading kernel function type from the model file!\n");
216 }
217 (*param).KERNEL = tempInt;
218
219 // kernel parameter (GAMMA) parameter
220 fseek (fModel, strlen("KERNEL_GAMMA_PARAM: "), SEEK_CUR);
221 if (!fscanf(fModel, "%f\n", &tempFloat))
222 {
223     svmPrintErrorString("Error reading RBF kernel width parameter from the model file!\n");
224 }
225 (*param).KERNEL_GAMMA_PARAM = tempFloat;
226
227 // kernel degree/slope parameter
228 fseek (fModel, strlen("KERNEL_DEGREE_PARAM: "), SEEK_CUR);
229 if (!fscanf(fModel, "%f\n", &tempFloat))
230 {
231     svmPrintErrorString("Error reading kernel degree/slope parameter from the model file!\n");
232 }
233 (*param).KERNEL_DEGREE_PARAM = tempFloat;
234
235 // kernel coefficient/intercept parameter
236 fseek (fModel, strlen("KERNEL_COEF_PARAM: "), SEEK_CUR);
237 if (!fscanf(fModel, "%f\n", &tempFloat))
238 {
239     svmPrintErrorString("Error reading kernel coefficient/intercept parameter from the model
file!\n");
240 }
241 (*param).KERNEL_COEF_PARAM = tempFloat;
242
243 // load the model
244 fseek (fModel, strlen("MODEL:\n") + 1, SEEK_CUR);
245
246 for (i = 0; i < numWeights[0]; i++) // for every weight
247 {
248     budgetedVectorBSGD *eNew = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT,
numClasses);
249     sqrNorm = 0.0L;
250
251     // get alphas and features
252     doneReadingBool = false;
253     while (!doneReadingBool)
254     {
255         doneReadingBool = fgetWord(fModel, oneWord);
256         if (strlen(oneWord) == 0)
257             continue;
258
259         text = oneWord;
260         if ((pos = (int) text.find(":"))
261         {
262             tempInt = atoi(text.substr(0, pos).c_str());
263             tempFloat = (float) atof(text.substr(pos + 1, text.length()).c_str());
264
265             // alphas have negative index, features have positive
266             if (tempInt > 0)
267             {
268                 (*eNew)[tempInt - 1] = tempFloat;
269                 sqrNorm += (long double) (tempFloat * tempFloat);
270             }
271             else
272                 (*eNew).alphas[- tempInt - 1] = tempFloat;
273         }
274     }
275     eNew->setSqrL2norm(sqrNorm);
276
277     (*modelBSGD).push_back(eNew);
278     eNew = NULL;
279 }
280
281 fclose(fModel);
282 return true;
283 }

```

5.5.2.3 saveToTextFile()

```

bool budgetedModelBSGD::saveToTextFile (
    const char * filename,

```

```
vector< int > * yLabels,
parameters * param ) [virtual]
```

Saves the trained BSGD model to .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.

Returns

Returns false if error encountered, otherwise true.

The text file has the following rows: [ALGORITHM, DIMENSION, NUMBER_OF_CLASSES, LABELS, NUMBER_OF_WEIGHTS, BIAS_TERM, KERNEL_WIDTH, MODEL]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

Each row corresponds to one support vector (or weight). The first elements of each weight correspond to alpha parameters for each class, given in order of "labels" member of the Matlab structure. However, since alpha can be equal to 0, we use LIBSVM format to store alphas, as -class_index:class-specific_alpha, where we added '-' (minus sign) in front of the class index to differentiate between class indices and feature indices that follow. After the alphas, in the same row the elements of the weights (or support vectors) for each feature are given in LIBSVM format.

Implements [budgetedModel](#).

Definition at line 60 of file bsgd.cpp.

```
61 {
62     unsigned int i, j;
63     FILE *fModel = NULL;
64     fModel = fopen(filename, "wt");
65     bool tempBool;
66
67     if (!fModel)
68         return false;
69
70     // algorithm
71     fprintf(fModel, "ALGORITHM: %d\n", (*param).ALGORITHM);
72
73     // dimension
74     fprintf(fModel, "DIMENSION: %d\n", (*param).DIMENSION);
75
76     // number of classes
77     fprintf(fModel, "NUMBER_OF_CLASSES: %d\n", (int) (*yLabels).size());
78
79     // labels
80     fprintf(fModel, "LABELS:");
81     for (i = 0; i < (*yLabels).size(); i++)
82         fprintf(fModel, " %d", (*yLabels)[i]);
83     fprintf(fModel, "\n");
84
85     // number of weights
86     fprintf(fModel, "NUMBER_OF_WEIGHTS: ");
87     fprintf(fModel, "%d\n", (int) (*modelBSGD).size());
88
89     // bias parameter
90     fprintf(fModel, "BIAS_TERM: %f\n", (*param).BIAS_TERM);
91
92     // kernel function
93     fprintf(fModel, "KERNEL_FUNCTION: %d\n", (*param).KERNEL);
94
95     // kernel width (GAMMA) parameter
96     fprintf(fModel, "KERNEL_GAMMA_PARAM: %f\n", (*param).KERNEL_GAMMA_PARAM);
97
98     // kernel degree/slope parameter
99     fprintf(fModel, "KERNEL_DEGREE_PARAM: %f\n", (*param).KERNEL_DEGREE_PARAM);
100
101     // kernel coef/intercept parameter
102     fprintf(fModel, "KERNEL_COEF_PARAM: %f\n", (*param).KERNEL_COEF_PARAM);
103 }
```

```

104 // save the model
105 fprintf(fModel, "MODEL:\n");
106 for (i = 0; i < (*modelBSGD).size(); i++)
107 //for (i = 0; i < 50; i++)
108 {
109     for (j = 0; j < (*yLabels).size(); j++)
110     {
111         // alphas have negative index to differentiate them from features
112         if ((*modelBSGD)[i]).alphas[j] != 0.0)
113             fprintf(fModel, "-d:%2.10f ", j + 1, (double) ((*modelBSGD)[i]).alphas[j]);
114     }
115
116     // this tempBool is used so that the line doesn't end with a white-space, it makes our life
117     // easier when reading word-by-word from the model file using fgetWord(); we can, of course,
118     // do without it, but to avoid unnecessary checks during loading of the model we do it here
119     tempBool = true;
120     for (j = 0; j < (*param).DIMENSION; j++) // for every feature
121     {
122         if ((*modelBSGD)[i])[j] != 0.0)
123         {
124             if (tempBool)
125             {
126                 fprintf(fModel, "d:%2.10f", j + 1, ((*modelBSGD)[i])[j]);
127                 tempBool = false;
128             }
129             else
130                 fprintf(fModel, " %d:%2.10f", j + 1, ((*modelBSGD)[i])[j]);
131         }
132     }
133     fprintf(fModel, "\n");
134 }
135 fclose(fModel);
137 return true;
138 }

```

The documentation for this class was generated from the following files:

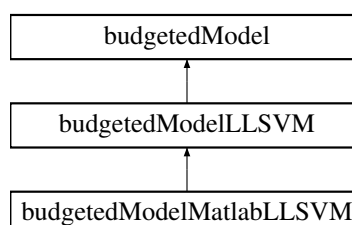
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.cpp

5.6 budgetedModelLLSVM Class Reference

Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to text file.

```
#include <llsvm.h>
```

Inheritance diagram for budgetedModelLLSVM:



Public Member Functions

- void [extendDimensionalityOfModel](#) (unsigned int newDim, [parameters](#) *param)
Extends the dimensionality of each landmark point in the LLSVM model.
- [budgetedModelLLSVM](#) (void)

Constructor, initializes the LLSVM model. Simply allocates memory for a vector of landmark points, where each is stored in [budgetedVectorLLSVM](#).

- [~budgetedModelLLSVM](#) (void)

Destructor, cleans up memory taken by LLSVM.

- bool [saveToTextFile](#) (const char *filename, vector< int > *yLabels, [parameters](#) *param)

Saves the trained LLSVM model to .txt file.

- bool [loadFromTextFile](#) (const char *filename, vector< int > *yLabels, [parameters](#) *param)

Loads the trained LLSVM model from .txt file.

Public Attributes

- vector< [budgetedVectorLLSVM](#) * > * [modelLLSVMlandmarks](#)

Holds landmark points, used to compute the transformation matrix [modelLLSVMmatrixW](#).

- VectorXd [modelLLSVMweightVector](#)

Holds weight vector, the solution of linear SVM on transformed points.

- MatrixXd [modelLLSVMmatrixW](#)

Holds transformation matrix, used to compute the mapping from original feature space into low-D space.

Additional Inherited Members

5.6.1 Detailed Description

Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to text file.

Definition at line 65 of file `llsvm.h`.

5.6.2 Member Function Documentation

5.6.2.1 extendDimensionalityOfModel()

```
void budgetedModelLLSVM::extendDimensionalityOfModel (
    unsigned int newDim,
    parameters * param ) [inline], [virtual]
```

Extends the dimensionality of each landmark point in the LLSVM model.

Extends the dimensionality of each landmark point in the LLSVM model. Called after new data chunk has been loaded, could be needed when user set the dimensionality of the data incorrectly, and we infer this important parameter during loading of the data.

Reimplemented from [budgetedModel](#).

Definition at line 86 of file `llsvm.h`.

```
87     {
88         // extend the dimensionality of each weight vector
89         for (unsigned int i = 0; i < (*modelLLSVMlandmarks).size(); i++)
90             (*modelLLSVMlandmarks)[i]->extendDimensionality(newDim, param);
91     };
```

5.6.2.2 loadFromTextFile()

```
bool budgetedModelLLSVM::loadFromTextFile (
    const char * filename,
    vector< int > * yLabels,
    parameters * param ) [virtual]
```

Loads the trained LLSVM model from .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
out	<i>yLabels</i>	Vector of possible labels.
out	<i>param</i>	The parameters of the algorithm.

Returns

Returns false if error encountered, otherwise true.

The text file has the following rows: [ALGORITHM, DIMENSION, NUMBER_OF_CLASSES, LABELS, NUMBER_OF_WEIGHTS, BIAS_TERM, KERNEL_WIDTH, MODEL]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

Each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space of the data set, stored in LIBSVM format.

Implements [budgetedModel](#).

Definition at line 783 of file llsvm.cpp.

```
784 {
785     unsigned int i, numClasses;
786     float tempFloat;
787     string text;
788     char oneWord[1024];
789     int pos, tempInt;
790     FILE *fModel = NULL;
791     bool doneReadingBool;
792     long double sqrNorm;
793
794     fModel = fopen(filename, "rt");
795     if (!fModel)
796         return false;
797
798     // algorithm
799     fseek(fModel, strlen("ALGORITHM: "), SEEK_CUR);
800     if (!fscanf(fModel, "%d\n", &((*param).ALGORITHM))
801     {
802         svmPrintErrorString("Error reading algorithm type from the model file!\n");
803     }
804
805     // dimension
806     fseek(fModel, strlen("DIMENSION:"), SEEK_CUR);
807     if (!fscanf(fModel, "%d\n", &((*param).DIMENSION))
808     {
809         svmPrintErrorString("Error reading number of dimensions from the model file!\n");
810     }
811
812     // number of classes (for LLSVM always equal to 2)
813     fseek(fModel, strlen("NUMBER_OF_CLASSES: "), SEEK_CUR);
814     if (!fscanf(fModel, "%d\n", &numClasses))
815     {
816         svmPrintErrorString("Error reading number of classes from the model file!\n");
817     }
818
819     // labels
820     fseek(fModel, strlen("LABELS: "), SEEK_CUR);
821     for (i = 0; i < numClasses; i++)
822     {
```

```

823         if (!fscanf(fModel, "%d ", &tempInt))
824         {
825             svmPrintErrorString("Error reading labels from the model file!\n");
826         }
827         (*yLabels).push_back(tempInt);
828     }
829
830     // number of weights
831     fseek (fModel, strlen("NUMBER_OF_WEIGHTS: "), SEEK_CUR);
832     if (!fscanf(fModel, "%d\n", &tempInt))
833     {
834         svmPrintErrorString("Error reading number of weights from the model file!\n");
835     }
836     (*param).BUDGET_SIZE = tempInt;
837
838     // bias parameter
839     fseek (fModel, strlen("BIAS_TERM: "), SEEK_CUR);
840     if (!fscanf(fModel, "%f\n", &tempFloat))
841     {
842         svmPrintErrorString("Error reading bias term from the model file!\n");
843     }
844     (*param).BIAS_TERM = tempFloat;
845
846     // read kernel function
847     fseek (fModel, strlen("KERNEL_FUNCTION: "), SEEK_CUR);
848     if (!fscanf(fModel, "%d\n", &tempInt))
849     {
850         svmPrintErrorString("Error reading kernel function type from the model file!\n");
851     }
852     (*param).KERNEL = tempInt;
853
854     // kernel parameter (GAMMA) parameter
855     fseek (fModel, strlen("KERNEL_GAMMA_PARAM: "), SEEK_CUR);
856     if (!fscanf(fModel, "%f\n", &tempFloat))
857     {
858         svmPrintErrorString("Error reading RBF kernel width parameter from the model file!\n");
859     }
860     (*param).KERNEL_GAMMA_PARAM = tempFloat;
861
862     // kernel degree/slope parameter
863     fseek (fModel, strlen("KERNEL_DEGREE_PARAM: "), SEEK_CUR);
864     if (!fscanf(fModel, "%f\n", &tempFloat))
865     {
866         svmPrintErrorString("Error reading kernel degree/slope parameter from the model file!\n");
867     }
868     (*param).KERNEL_DEGREE_PARAM = tempFloat;
869
870     // kernel coefficient/intercept parameter
871     fseek (fModel, strlen("KERNEL_COEF_PARAM: "), SEEK_CUR);
872     if (!fscanf(fModel, "%f\n", &tempFloat))
873     {
874         svmPrintErrorString("Error reading kernel coefficient/intercept parameter from the model
file!\n");
875     }
876     (*param).KERNEL_COEF_PARAM = tempFloat;
877
878     // allocate memory for model
879     modelLLSVMmatrixW.resize((*param).BUDGET_SIZE, (*param).BUDGET_SIZE);
880
881     // initialize weight (i.e., hyperplane) in the projected space to zero-vector
882     modelLLSVMweightVector.resize((*param).BUDGET_SIZE);
883
884     // load the model
885     fseek (fModel, strlen("MODEL:\n") + 1, SEEK_CUR);
886     for (i = 0; i < (*param).BUDGET_SIZE; i++)
887     {
888         budgetedVectorLLSVM *eNew = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
889         sqrNorm = 0.0L;
890
891         // get alphas and features below
892
893         // get linear SVM feature
894         doneReadingBool = fgetWord(fModel, oneWord);
895         modelLLSVMweightVector(i) = (double) atof(oneWord);
896
897         // get elements of modelLLSVMmatrixW
898         for (unsigned int j = 0; j < (*param).BUDGET_SIZE; j++)
899         {
900             doneReadingBool = fgetWord(fModel, oneWord);
901             modelLLSVMmatrixW(i, j) = (double) atof(oneWord);
902         }
903
904         // get features
905         while (!doneReadingBool)
906         {
907             doneReadingBool = fgetWord(fModel, oneWord);
908             if (strlen(oneWord) == 0)

```

```

909             continue;
910
911         text = oneWord;
912         if ((pos = (int) text.find(":"))
913         {
914             tempInt = atoi(text.substr(0, pos).c_str());
915             tempFloat = (float) atof(text.substr(pos + 1, text.length()).c_str());
916             (*eNew)[tempInt - 1] = tempFloat;
917             sqrNorm += (long double)(tempFloat * tempFloat);
918         }
919     }
920     eNew->setSqrL2norm(sqrNorm);
921     (*modelLLSVMlandmarks).push_back(eNew);
922     eNew = NULL;
923 }
924 fclose(fModel);
925 return true;
926 }

```

5.6.2.3 saveToTextFile()

```

bool budgetedModelLLSVM::saveToTextFile (
    const char * filename,
    vector< int > * yLabels,
    parameters * param ) [virtual]

```

Saves the trained LLSVM model to .txt file.

Parameters

in	<i>filename</i>	Filename of the .txt file where the model is saved.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.

Returns

Returns false if error encountered, otherwise true.

The text file has the following rows: [ALGORITHM, DIMENSION, NUMBER_OF_CLASSES, LABELS, NUMBER_OF_WEIGHTS, BIAS_TERM, KERNEL_WIDTH, MODEL]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

Each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space of the data set, stored in LIBSVM format.

Implements [budgetedModel](#).

Definition at line 709 of file llsvm.cpp.

```

710 {
711     unsigned int i, j;
712     FILE *fModel = NULL;
713
714     fModel = fopen(filename, "wt");
715     if (!fModel)
716         return false;
717
718     // algorithm
719     fprintf(fModel, "ALGORITHM: %d\n", (*param).ALGORITHM);
720
721     // dimension
722     fprintf(fModel, "DIMENSION: %d\n", (*param).DIMENSION);

```

```

723
724 // number of classes
725 fprintf(fModel, "NUMBER_OF_CLASSES: %d\n", (int) (*yLabels).size());
726
727 // labels
728 fprintf(fModel, "LABELS:");
729 for (i = 0; i < (*yLabels).size(); i++)
730     fprintf(fModel, " %d", (*yLabels)[i]);
731 fprintf(fModel, "\n");
732
733 // number of weights
734 fprintf(fModel, "NUMBER_OF_WEIGHTS:");
735 fprintf(fModel, " %d\n", (int) (*modelLLSVMlandmarks).size());
736
737 // bias parameter
738 fprintf(fModel, "BIAS_TERM: %f\n", (*param).BIAS_TERM);
739
740 // kernel function
741 fprintf(fModel, "KERNEL_FUNCTION: %d\n", (*param).KERNEL);
742
743 // kernel width (GAMMA) parameter
744 fprintf(fModel, "KERNEL_GAMMA_PARAM: %f\n", (*param).KERNEL_GAMMA_PARAM);
745
746 // kernel degree/slope parameter
747 fprintf(fModel, "KERNEL_DEGREE_PARAM: %f\n", (*param).KERNEL_DEGREE_PARAM);
748
749 // kernel coef/intercept parameter
750 fprintf(fModel, "KERNEL_COEF_PARAM: %f\n", (*param).KERNEL_COEF_PARAM);
751
752 // save the model
753 fprintf(fModel, "MODEL:\n");
754 for (i = 0; i < (*modelLLSVMlandmarks).size(); i++)
755 {
756     // put the i-th value of linear SVM hyperplane
757     fprintf(fModel, "%2.6f", (double)modelLLSVMweightVector(i));
758
759     // next, put the values of one row of modelLLSVMmatrixW
760     for (j = 0; j < (*param).BUDGET_SIZE; j++)
761         fprintf(fModel, " %2.6f", modelLLSVMmatrixW(i, j));
762
763     // finally, store the landmark point
764     for (j = 0; j < (*param).DIMENSION; j++)
765     {
766         if ((*modelLLSVMlandmarks)[i])[j] != 0.0)
767             fprintf(fModel, " %d:%2.6f", j + 1, ((*modelLLSVMlandmarks)[i])[j]);
768     }
769     fprintf(fModel, "\n");
770 }
771
772 fclose(fModel);
773 return true;
774 }

```

The documentation for this class was generated from the following files:

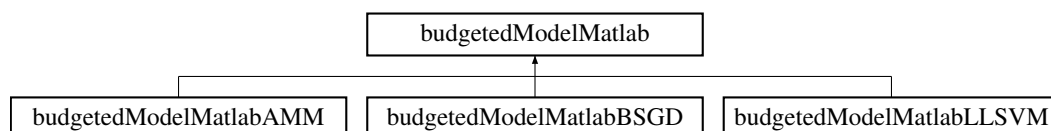
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/llsvm.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/llsvm.cpp

5.7 budgetedModelMatlab Class Reference

Interface which defines methods to load model from and save model to Matlab environment.

```
#include <budgetedSVM_matlab.h>
```

Inheritance diagram for budgetedModelMatlab:



Public Member Functions

- virtual void [saveToMatlabStruct](#) (mxAarray *plhs[], vector< int > *yLabels, [parameters](#) *param)=0
Save the trained model to Matlab, by creating Matlab structure.
- virtual bool [loadFromMatlabStruct](#) (const mxArray *matlabStruct, vector< int > *yLabels, [parameters](#) *param, const char **msg)=0
Loads the trained model from Matlab structure.

Static Public Member Functions

- static int [getAlgorithm](#) (const mxArray *matlabStruct)
Get algorithm from the trained model stored in Matlab structure.

5.7.1 Detailed Description

Interface which defines methods to load model from and save model to Matlab environment.

Definition at line 73 of file budgetedSVM_matlab.h.

5.7.2 Member Function Documentation

5.7.2.1 getAlgorithm()

```
static int budgetedModelMatlab::getAlgorithm (
    const mxArray * matlabStruct ) [static]
```

Get algorithm from the trained model stored in Matlab structure.

Parameters

in	<i>matlabStruct</i>	Pointer to Matlab structure.
----	---------------------	------------------------------

Returns

-1 if error, otherwise returns algorithm code from the model file.

Definition at line 67 of file budgetedSVM_matlab.cpp.

```
68 {
69     if (mxGetNumberOfFields(matlabStruct) != NUM_OF_RETURN_FIELD)
70         return -1;
71
72     // get algorithm
73     return (int) (* (mxGetPr (mxGetFieldByNumber (matlabStruct, 0, 0))));
74 }
```

5.7.2.2 loadFromMatlabStruct()

```
bool budgetedModelMatlab::loadFromMatlabStruct (
    const mxArray * matlabStruct,
    vector< int > * yLabels,
    parameters * param,
    const char ** msg ) [pure virtual]
```

Loads the trained model from Matlab structure.

Parameters

in	<i>matlabStruct</i>	Pointer to Matlab structure.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.
out	<i>msg</i>	Error message, if error encountered.

Returns

False if error encountered, otherwise true.

The Matlab structure is organized as [*algorithm*, *dimension*, *numClasses*, *labels*, *numWeights*, *paramBias*, *kernel*, *Width*, *model*]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

- AMM online, AMM batch, and Pegasos: The model is stored as $((dimension + 1) \times numWeights)$ matrix. The first element of each weight is the degradation of the weight, followed by values of the weight for each feature of the data set. If bias term is non-zero, then the final element of each weight corresponds to bias term, and the matrix is of size $((dimension + 2) \times numWeights)$. By looking at *labels* and *numWeights* members of Matlab structure we can find out which weights belong to which class. For example, first *numWeights*[0] weights belong to *labels*[0] class, next *numWeights*[1] weights belong to *labels*[1] class, and so on.
- BSGD: The model is stored as $((numClasses + dimension) \times numWeights)$ matrix. The first *numClasses* elements of each weight correspond to alpha parameters for each class, given in order of "labels" member of the Matlab structure. This is followed by elements of the weights (or support vectors) for each feature of the data set.
- LLSVM: The model is stored as $((1 + dimension) \times numWeights)$ matrix. Each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space.

Implemented in [budgetedModelMatlabLLSVM](#), [budgetedModelMatlabBSGD](#), and [budgetedModelMatlabAMM](#).

5.7.2.3 saveToMatlabStruct()

```
void budgetedModelMatlab::saveToMatlabStruct (
    mxArray * plhs[],
    vector< int > * yLabels,
    parameters * param ) [pure virtual]
```

Save the trained model to Matlab, by creating Matlab structure.

Parameters

out	<i>plhs</i>	Pointer to Matlab output.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.

The Matlab structure is organized as `[algorithm, dimension, numClasses, labels, numWeights, paramBias, kernelWidth, mode]`. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

- AMM online, AMM batch, and Pegasos: The model is stored as $((dimension + 1) \times numWeights)$ matrix. The first element of each weight is the degradation of the weight, followed by values of the weight for each feature of the data set. If bias term is non-zero, then the final element of each weight corresponds to bias term, and the matrix is of size $((dimension + 2) \times numWeights)$. By looking at *labels* and *numWeights* members of Matlab structure we can find out which weights belong to which class. For example, first *numWeights*[0] weights belong to *labels*[0] class, next *numWeights*[1] weights belong to *labels*[1] class, and so on.
- BSGD: The model is stored as $((numClasses + dimension) \times numWeights)$ matrix. The first *numClasses* elements of each weight correspond to alpha parameters for each class, given in order of *labels* member of the Matlab structure. This is followed by elements of the weights (or support vectors) for each feature of the data set.
- LLSVM: The model is stored as $((1 + dimension) \times numWeights)$ matrix. Each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space.

Implemented in [budgetedModelMatlabLLSVM](#), [budgetedModelMatlabBSGD](#), and [budgetedModelMatlabAMM](#).

The documentation for this class was generated from the following files:

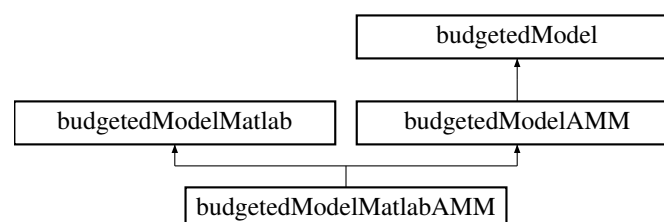
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.h](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.cpp](#)

5.8 budgetedModelMatlabAMM Class Reference

Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to Matlab environment.

```
#include <budgetedSVM_matlab.h>
```

Inheritance diagram for budgetedModelMatlabAMM:



Public Member Functions

- void `saveToMatlabStruct` (mxArray *plhs[], vector< int > *yLabels, `parameters` *param)
Save the trained model to Matlab, by creating Matlab structure.
- bool `loadFromMatlabStruct` (const mxArray *matlabStruct, vector< int > *yLabels, `parameters` *param, const char **msg)
Loads the trained model from Matlab structure.

Additional Inherited Members

5.8.1 Detailed Description

Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to Matlab environment.

Definition at line 137 of file budgetedSVM_matlab.h.

5.8.2 Member Function Documentation

5.8.2.1 loadFromMatlabStruct()

```
bool budgetedModelMatlabAMM::loadFromMatlabStruct (
    const mxArray * matlabStruct,
    vector< int > * yLabels,
    parameters * param,
    const char ** msg ) [virtual]
```

Loads the trained model from Matlab structure.

Parameters

in	<code>matlabStruct</code>	Pointer to Matlab structure.
in	<code>yLabels</code>	Vector of possible labels.
in	<code>param</code>	The parameters of the algorithm.
out	<code>msg</code>	Error message, if error encountered.

Returns

False if error encountered, otherwise true.

The Matlab structure is organized as ["algorithm", "dimension", "numClasses", "labels", "numWeights", "paramBias", "kernelWidth", "model"]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

The model is stored as (("dimension" + 1) by "numWeights") matrix. The first element of each weight is the degradation of the weight, followed by values of the weight for each feature of the data set. If bias term is non-zero, then

the final element of each weight corresponds to bias term, and the matrix is of size $((\text{"dimension"} + 2) \times \text{"numWeights"})$. By looking at "labels" and "numWeights" members of Matlab structure we can find out which weights belong to which class. For example, first numWeights[0] weights belong to labels[0] class, next numWeights[1] weights belong to labels[1] class, and so on.

Implements [budgetedModelMatlab](#).

Definition at line 425 of file budgetedSVM_matlab.cpp.

```

426 {
427     int i, j, numOfFields, numClasses, currClass, classCounter;
428     double *ptr;
429     int id = 0;
430     mxArray **rhs;
431     vector<unsigned int> numWeights;
432     double sqrNorm;
433
434     numOfFields = mxGetNumberOfFields(matlabStruct);
435     if (numOfFields != NUM_OF_RETURN_FIELD)
436     {
437         *msg = "number of return fields is not correct";
438         return false;
439     }
440     rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * numOfFields);
441
442     for (i = 0; i < numOfFields; i++)
443         rhs[i] = mxGetFieldByNumber(matlabStruct, 0, i);
444
445     // algorithm
446     ptr = mxGetPr(rhs[id]);
447     param->ALGORITHM = (unsigned int)ptr[0];
448     id++;
449
450     // dimension
451     ptr = mxGetPr(rhs[id]);
452     param->DIMENSION = (unsigned int)ptr[0];
453     id++;
454
455     // numClasses
456     ptr = mxGetPr(rhs[id]);
457     numClasses = (unsigned int)ptr[0];
458     id++;
459
460     // labels
461     if (mxIsEmpty(rhs[id]) == 0)
462     {
463         ptr = mxGetPr(rhs[id]);
464         for(i = 0; i < numClasses; i++)
465         {
466             (*yLabels).push_back((int)ptr[i]);
467
468             // add to model empty weight vector for each class
469             vector<budgetedVectorAMM*> tempV;
470             (*modelMM).push_back(tempV);
471         }
472     }
473     id++;
474
475     // numWeights
476     if (mxIsEmpty(rhs[id]) == 0)
477     {
478         ptr = mxGetPr(rhs[id]);
479         for(i = 0; i < numClasses; i++)
480         {
481             numWeights.push_back((int)ptr[i]);
482         }
483     }
484     id++;
485
486     // bias term
487     ptr = mxGetPr(rhs[id]);
488     param->BIAS_TERM = (double)ptr[0];
489     id++;
490
491     // kernel choice, just skip
492     id++;
493
494     // kernel width gamma
495     id++;
496
497     // kernel degree/slope param
498     id++;
499
500     // kernel intercept param
501     id++;

```

```

502
503 // weights
504 int sr, sc;
505 mwIndex *ir, *jc;
506
507 sr = (int)mxGetN(rhs[id]);
508 sc = (int)mxGetM(rhs[id]);
509
510 ptr = mxGetPr(rhs[id]);
511 ir = mxGetIr(rhs[id]);
512 jc = mxGetJc(rhs[id]);
513
514 // weights are in columns
515 currClass = classCounter = 0;
516 for (i = 0; i < sr; i++)
517 {
518     int low = (int)jc[i], high = (int)jc[i + 1];
519     budgetedVectorAMM *eNew = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
520     sqrNorm = 0.0;
521
522     for (j = low; j < high; j++)
523     {
524         if (param->ALGORITHM == PEGASOS)
525             ((*eNew)[(int)ir[j]]) = (float)ptr[j];
526         else if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
527         {
528             if (j == low)
529                 eNew->setDegradation(ptr[j]);
530             else
531             {
532                 ((*eNew)[(int)ir[j] - 1]) = (float)ptr[j];
533                 sqrNorm += (ptr[j] * ptr[j]);
534             }
535         }
536     }
537     eNew->setSqrL2norm(sqrNorm);
538     (*modelMM)[currClass].push_back(eNew);
539     eNew = NULL;
540
541     // increment weight counter and check if new class is starting
542     if (++classCounter == numWeights[currClass])
543     {
544         classCounter = 0;
545         currClass++;
546     }
547 }
548 id++;
549
550 mxFree(rhs);
551 return true;
552 }

```

5.8.2.2 saveToMatlabStruct()

```

void budgetedModelMatlabAMM::saveToMatlabStruct (
    mxArray * plhs[],
    vector< int > * yLabels,
    parameters * param ) [virtual]

```

Save the trained model to Matlab, by creating Matlab structure.

Parameters

out	<i>plhs</i>	Pointer to Matlab output.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.

The Matlab structure is organized as ["algorithm", "dimension", "numClasses", "labels", "numWeights", "paramBias", "kernelWidth", "model"]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

The model is stored as ("dimension" + 1) by "numWeights") matrix. The first element of each weight is the degradation of the weight, followed by values of the weight for each feature of the data set. If bias term is non-zero, then the final element of each weight corresponds to bias term, and the matrix is of size ("dimension" + 2) by "numWeights". By looking at "labels" and "numWeights" members of Matlab structure we can find out which weights belong to which class. For example, first numWeights[0] weights belong to labels[0] class, next numWeights[1] weights belong to labels[1] class, and so on.

Implements [budgetedModelMatlab](#).

Definition at line 272 of file budgetedSVM_matlab.cpp.

```

273 {
274     unsigned int i, j, numWeights = 0, cnt;
275     double *ptr;
276     mxArray *returnModel, **rhs;
277     int outID = 0;
278
279     rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * NUM_OF_RETURN_FIELD);
280
281     // algorithm type
282     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
283     ptr = mxGetPr(rhs[outID]);
284     ptr[0] = param->ALGORITHM;
285     outID++;
286
287     // dimension
288     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
289     ptr = mxGetPr(rhs[outID]);
290     ptr[0] = param->DIMENSION;
291     outID++;
292
293     // number of classes
294     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
295     ptr = mxGetPr(rhs[outID]);
296     ptr[0] = (double) (*yLabels).size();
297     outID++;
298
299     // labels
300     rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
301     ptr = mxGetPr(rhs[outID]);
302     for (i = 0; i < (*yLabels).size(); i++)
303         ptr[i] = (*yLabels)[i];
304     outID++;
305
306     // total number of weights
307     rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
308     ptr = mxGetPr(rhs[outID]);
309     for (i = 0; i < (*modelMM).size(); i++)
310     {
311         ptr[i] = (double) (*modelMM)[i].size();
312         numWeights += (unsigned int) (*modelMM)[i].size();
313     }
314     outID++;
315
316     // bias param
317     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
318     ptr = mxGetPr(rhs[outID]);
319     ptr[0] = param->BIAS_TERM;
320     outID++;
321
322     // kernel choice
323     rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
324     outID++;
325
326     // kernel width gamma
327     rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
328     outID++;
329
330     // kernel degree/slope param
331     rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
332     outID++;
333
334     // kernel intercept param
335     rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
336     outID++;
337
338     // weights
339     int irIndex, nonZeroElement;
340     mwIndex *ir, *jc;
341
342     // find how many non-zero elements there are
343     nonZeroElement = 0;
344     for (i = 0; i < (*modelMM).size(); i++)
345     {

```

```

346         for (j = 0; j < (*modelMM)[i].size(); j++)
347         {
348             for (unsigned int k = 0; k < (*param).DIMENSION; k++)           // for every feature
349             {
350                 if ((*(*modelMM)[i][j]))[k] != 0.0)
351                     nonZeroElement++;
352             }
353         }
354     }
355
356     // +1 is for degradation of AMM algorithms, it will be the first number in the row representing a
weight
357     if (param->ALGORITHM == PEGASOS)
358         rhs[outID] = mxCreateSparse(param->DIMENSION, numWeights, nonZeroElement, mxREAL);
359     else if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
360         rhs[outID] = mxCreateSparse(param->DIMENSION + 1, numWeights, nonZeroElement + numWeights,
mxREAL);
361     ir = mxGetIr(rhs[outID]);
362     jc = mxGetJc(rhs[outID]);
363     ptr = mxGetPr(rhs[outID]);
364     jc[0] = irIndex = cnt = 0;
365     for (i = 0; i < (*modelMM).size(); i++)
366     {
367         for (j = 0; j < (*modelMM)[i].size(); j++)
368         {
369             int xIndex = 0;
370
371             // this adds degradation to the beginning of a vector, more compact
372             if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
373             {
374                 ir[irIndex] = 0;
375                 ptr[irIndex] = (*modelMM)[i][j]->getDegradation();
376                 irIndex++, xIndex++;
377             }
378
379             // add the actual features
380             for (unsigned int k = 0; k < (*param).DIMENSION; k++)           // for every feature
381             {
382                 if ((*(*modelMM)[i][j]))[k] != 0.0)
383                 {
384                     if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
385                         ir[irIndex] = k + 1;
386                     else if (param->ALGORITHM == PEGASOS)
387                         ir[irIndex] = k;
388                     ptr[irIndex] = ((*modelMM)[i][j])[k];
389                     irIndex++, xIndex++;
390                 }
391             }
392             jc[cnt + 1] = jc[cnt] + xIndex;
393             cnt++;
394         }
395     }
396     // commented, since now it is appended to the weight matrix
397     /// degradations
398     cnt = 0;
399     rhs[outID] = mxCreateDoubleMatrix(numWeights, 1, mxREAL);
400     ptr = mxGetPr(rhs[outID]);
401     for (i = 0; i < (*modelMM).size(); i++)
402         for (j = 0; j < (*modelMM)[i].size(); j++)
403             ptr[cnt++] = (*modelMM)[i][j]->degradation;
404     outID++;*/
405
406     /* Create a struct matrix contains NUM_OF_RETURN_FIELD fields */
407     returnModel = mxCreateStructMatrix(1, 1, NUM_OF_RETURN_FIELD, fieldNames);
408
409     /* Fill struct matrix with input arguments */
410     for(i = 0; i < NUM_OF_RETURN_FIELD; i++)
411         mxSetField(returnModel, 0, fieldNames[i], mxDuplicateArray(rhs[i]));
412
413     plhs[0] = returnModel;
414     mxFree(rhs);
415 }

```

The documentation for this class was generated from the following files:

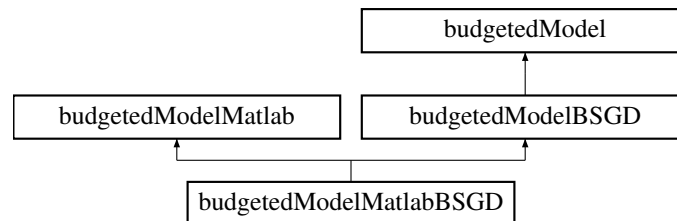
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.h](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.cpp](#)

5.9 budgetedModelMatlabBSGD Class Reference

Class which holds the BSGD model, and implements methods to load BSGD model from and save BSGD model to Matlab environment.

```
#include <budgetedSVM_matlab.h>
```

Inheritance diagram for budgetedModelMatlabBSGD:



Public Member Functions

- void [saveToMatlabStruct](#) (mxArray *plhs[], vector< int > *yLabels, [parameters](#) *param)
Save the trained model to Matlab, by creating Matlab structure.
- bool [loadFromMatlabStruct](#) (const mxArray *matlabStruct, vector< int > *yLabels, [parameters](#) *param, const char **msg)
Loads the trained model from Matlab structure.

Additional Inherited Members

5.9.1 Detailed Description

Class which holds the BSGD model, and implements methods to load BSGD model from and save BSGD model to Matlab environment.

Definition at line 178 of file budgetedSVM_matlab.h.

5.9.2 Member Function Documentation

5.9.2.1 loadFromMatlabStruct()

```
bool budgetedModelMatlabBSGD::loadFromMatlabStruct (
    const mxArray * matlabStruct,
    vector< int > * yLabels,
    parameters * param,
    const char ** msg ) [virtual]
```

Loads the trained model from Matlab structure.

Parameters

in	<i>matlabStruct</i>	Pointer to Matlab structure.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.
out	<i>msg</i>	Error message, if error encountered.

Returns

False if error encountered, otherwise true.

The Matlab structure is organized as ["algorithm", "dimension", "numClasses", "labels", "numWeights", "paramBias", "kernelWidth", "model"]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

The model is stored as $(("numClasses" + "dimension") \times "numWeights")$ matrix. The first "numClasses" elements of each weight correspond to alpha parameters for each class, given in order of "labels" member of the Matlab structure. This is followed by elements of the weights (or support vectors) for each feature of the data set.

Implements [budgetedModelMatlab](#).

Definition at line 708 of file budgetedSVM_matlab.cpp.

```

709 {
710     int i, j, numOffFields, numClasses, currClass, classCounter;
711     double *ptr;
712     int id = 0;
713     mxArray **rhs;
714     vector<unsigned int> numWeights;
715     double sqrNorm;
716
717     numOffFields = mxGetNumberOfFields(matlabStruct);
718     if (numOffFields != NUM_OF_RETURN_FIELD)
719     {
720         *msg = "number of return fields is not correct";
721         return false;
722     }
723     rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * numOffFields);
724
725     for (i = 0; i < numOffFields; i++)
726         rhs[i] = mxGetFieldByNumber(matlabStruct, 0, i);
727
728     // algorithm
729     ptr = mxGetPr(rhs[id]);
730     param->ALGORITHM = (unsigned int)ptr[0];
731     id++;
732
733     // dimension
734     ptr = mxGetPr(rhs[id]);
735     param->DIMENSION = (unsigned int)ptr[0];
736     id++;
737
738     // numClasses
739     ptr = mxGetPr(rhs[id]);
740     numClasses = (unsigned int)ptr[0];
741     id++;
742
743     // labels
744     if (mxIsEmpty(rhs[id]) == 0)
745     {
746         ptr = mxGetPr(rhs[id]);
747         for (i = 0; i < numClasses; i++)
748             (*yLabels).push_back((int)ptr[i]);
749     }
750     id++;
751
752     // numWeights, just skip
753     id++;
754
755     // bias term
756     ptr = mxGetPr(rhs[id]);
757     param->BIAS_TERM = (double)ptr[0];
758     id++;
759 }

```

```

760 // kernel choice, just skip
761 ptr = mxGetPr(rhs[id]);
762 param->KERNEL = (unsigned int) ptr[0];
763 id++;
764
765 // kernel width gamma
766 ptr = mxGetPr(rhs[id]);
767 param->KERNEL_GAMMA_PARAM = (double)ptr[0];
768 id++;
769
770 // kernel degree/slope param
771 ptr = mxGetPr(rhs[id]);
772 param->KERNEL_DEGREE_PARAM = (double)ptr[0];
773 id++;
774
775 // kernel intercept param
776 ptr = mxGetPr(rhs[id]);
777 param->KERNEL_COEF_PARAM = (double)ptr[0];
778 id++;
779
780 // weights
781 int sr, sc;
782 mwIndex *ir, *jc;
783
784 sr = (int)mxGetN(rhs[id]);
785 sc = (int)mxGetM(rhs[id]);
786
787 ptr = mxGetPr(rhs[id]);
788 ir = mxGetIr(rhs[id]);
789 jc = mxGetJc(rhs[id]);
790
791 // weights are in columns
792 currClass = classCounter = 0;
793 for (i = 0; i < sr; i++)
794 {
795     int low = (int)jc[i], high = (int)jc[i + 1];
796     budgetedVectorBSGD *eNew = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT,
numClasses);
797     sqrNorm = 0.0;
798
799     for (j = low; j < high; j++)
800     {
801         if ((unsigned int)ir[j] < (*yLabels).size())
802         {
803             // get alpha values
804             eNew->alphas[(int)ir[j]] = ptr[j];
805         }
806         else
807         {
808             // get features
809             ((*eNew)[(int)ir[j] - (int) (*yLabels).size()]) = (float)ptr[j];
810             sqrNorm += (ptr[j] * ptr[j]);
811         }
812     }
813     eNew->setSqrL2norm(sqrNorm);
814     (*modelBSGD).push_back(eNew);
815     eNew = NULL;
816 }
817 id++;
818
819 mxFree(rhs);
820 return true;
821 }

```

5.9.2.2 saveToMatlabStruct()

```

void budgetedModelMatlabBSGD::saveToMatlabStruct (
    mxArray * plhs[],
    vector< int > * yLabels,
    parameters * param ) [virtual]

```

Save the trained model to Matlab, by creating Matlab structure.

Parameters

out	<i>plhs</i>	Pointer to Matlab output.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.

The Matlab structure is organized as ["algorithm", "dimension", "numClasses", "labels", "numWeights", "paramBias", "kernelWidth", "model"]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

The model is stored as ("numClasses" + "dimension") by "numWeights") matrix. The first "numClasses" elements of each weight correspond to alpha parameters for each class, given in order of "labels" member of the Matlab structure. This is followed by elements of the weights (or support vectors) for each feature of the data set.

Implements [budgetedModelMatlab](#).

Definition at line 560 of file budgetedSVM_matlab.cpp.

```

561 {
562     unsigned int i, j, numWeights = 0, cnt;
563     double *ptr;
564     mxArray *returnModel, **rhs;
565     int outID = 0;
566
567     rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * NUM_OF_RETURN_FIELD);
568
569     // algorithm type
570     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
571     ptr = mxGetPr(rhs[outID]);
572     ptr[0] = param->ALGORITHM;
573     outID++;
574
575     // dimension
576     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
577     ptr = mxGetPr(rhs[outID]);
578     ptr[0] = param->DIMENSION;
579     outID++;
580
581     // number of classes
582     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
583     ptr = mxGetPr(rhs[outID]);
584     ptr[0] = (double) (*yLabels).size();
585     outID++;
586
587     // labels
588     rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
589     ptr = mxGetPr(rhs[outID]);
590     for (i = 0; i < (*yLabels).size(); i++)
591         ptr[i] = (*yLabels)[i];
592     outID++;
593
594     // total number of weights
595     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
596     ptr = mxGetPr(rhs[outID]);
597     ptr[0] = (double) (*modelBSGD).size();
598     numWeights = (unsigned int) (*modelBSGD).size();
599     outID++;
600
601     // bias param
602     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
603     ptr = mxGetPr(rhs[outID]);
604     ptr[0] = param->BIAS_TERM;
605     outID++;
606
607     // kernel choice
608     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
609     ptr = mxGetPr(rhs[outID]);
610     ptr[0] = param->KERNEL;
611     outID++;
612
613     // kernel width gamma
614     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
615     ptr = mxGetPr(rhs[outID]);
616     ptr[0] = param->KERNEL_GAMMA_PARAM;
617     outID++;
618
619     // kernel degree/slope param
620     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
621     ptr = mxGetPr(rhs[outID]);
622     ptr[0] = param->KERNEL_DEGREE_PARAM;
623     outID++;
624
625     // kernel intercept param
626     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
627     ptr = mxGetPr(rhs[outID]);
628     ptr[0] = param->KERNEL_COEF_PARAM;
629     outID++;
630
631     // weights, different for MM algorithms, BSGD and LLSVM

```

```

632     int irIndex, nonZeroElement;
633     mwIndex *ir, *jc;
634
635     // find how many non-zero elements there are
636     nonZeroElement = 0;
637     for (i = 0; i < (*modelBSGD).size(); i++)
638     {
639         // count non-zero features
640         for (j = 0; j < (*param).DIMENSION; j++)
641         {
642             if ((*(*modelBSGD)[i]))[j] != 0.0)
643                 nonZeroElement++;
644         }
645
646         // count non-zero alphas also
647         for (j = 0; j < (*yLabels).size(); j++)
648         {
649             if ((*(*modelBSGD)[i]).alphas[j] != 0.0)
650                 nonZeroElement++;
651         }
652     }
653
654     // +(*yLabels).size() is for the alpha parameters of each BSGD weight
655     rhs[outID] = mxCreateSparse(param->DIMENSION + (*yLabels).size(), numWeights, nonZeroElement,
mxREAL);
656     ir = mxGetIr(rhs[outID]);
657     jc = mxGetJc(rhs[outID]);
658     ptr = mxGetPr(rhs[outID]);
659     jc[0] = irIndex = cnt = 0;
660     for (i = 0; i < (*modelBSGD).size(); i++)
661     {
662         int xIndex = 0;
663
664         // this adds alpha weights to the beginning of a vector, more compact
665         for (j = 0; j < (*yLabels).size(); j++)
666         {
667             if ((*(*modelBSGD)[i]).alphas[j] != 0.0)
668             {
669                 ir[irIndex] = j;
670                 ptr[irIndex] = ((*(*modelBSGD)[i]).alphas[j];
671                 irIndex++, xIndex++;
672             }
673         }
674
675         // add the actual features
676         for (j = 0; j < (*param).DIMENSION; j++)           // for every feature
677         {
678             if ((*(*modelBSGD)[i]))[j] != 0.0)
679             {
680                 ir[irIndex] = j + (*yLabels).size();        // shift it to accomodate alpha weights
681                 ptr[irIndex] = ((*(*modelBSGD)[i]))[j];
682                 irIndex++, xIndex++;
683             }
684         }
685         jc[cnt + 1] = jc[cnt] + xIndex;
686         cnt++;
687     }
688
689     /* Create a struct matrix contains NUM_OF_RETURN_FIELD fields */
690     returnModel = mxCreateStructMatrix(1, 1, NUM_OF_RETURN_FIELD, fieldNames);
691
692     /* Fill struct matrix with input arguments */
693     for(i = 0; i < NUM_OF_RETURN_FIELD; i++)
694         mxSetField(returnModel, 0, fieldNames[i], mxDuplicateArray(rhs[i]));
695
696     plhs[0] = returnModel;
697     mxFree(rhs);
698 }

```

The documentation for this class was generated from the following files:

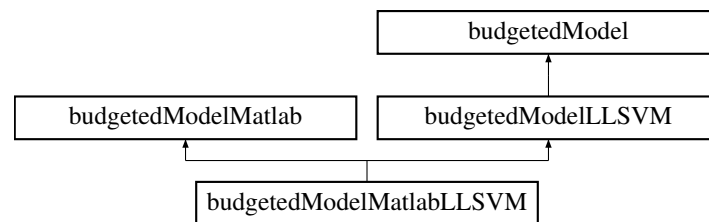
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.h](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/[budgetedSVM_matlab.cpp](#)

5.10 budgetedModelMatlabLLSVM Class Reference

Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to Matlab environment.

```
#include <budgetedSVM_matlab.h>
```

Inheritance diagram for budgetedModelMatlabLLSVM:



Public Member Functions

- void [saveToMatlabStruct](#) (mxAarray *plhs[], vector< int > *yLabels, [parameters](#) *param)
Save the trained model to Matlab, by creating Matlab structure.
- bool [loadFromMatlabStruct](#) (const mxArray *matlabStruct, vector< int > *yLabels, [parameters](#) *param, const char **msg)
Loads the trained model from Matlab structure.

Additional Inherited Members

5.10.1 Detailed Description

Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to Matlab environment.

Definition at line 215 of file budgetedSVM_matlab.h.

5.10.2 Member Function Documentation

5.10.2.1 loadFromMatlabStruct()

```
bool budgetedModelMatlabLLSVM::loadFromMatlabStruct (
    const mxArray * matlabStruct,
    vector< int > * yLabels,
    parameters * param,
    const char ** msg ) [virtual]
```

Loads the trained model from Matlab structure.

Parameters

in	<i>matlabStruct</i>	Pointer to Matlab structure.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.
out	<i>msg</i>	Error message, if error encountered.

Returns

False if error encountered, otherwise true.

The Matlab structure is organized as ["algorithm", "dimension", "numClasses", "labels", "numWeights", "paramBias", "kernelWidth", "model"]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

The model is stored as ((1 + "dimension") by "numWeights") matrix. Each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space.

Implements [budgetedModelMatlab](#).

Definition at line 978 of file budgetedSVM_matlab.cpp.

```

979 {
980     unsigned int i, j, numOfFields, numClasses;
981     double *ptr, sqrNorm;
982     int id = 0;
983     mxArray **rhs;
984
985     numOfFields = mxGetNumberOfFields(matlabStruct);
986     if (numOfFields != NUM_OF_RETURN_FIELD)
987     {
988         *msg = "Number of return fields is not correct.";
989         return false;
990     }
991     rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * numOfFields);
992
993     for (i = 0; i < numOfFields; i++)
994         rhs[i] = mxGetFieldByNumber(matlabStruct, 0, i);
995
996     // algorithm
997     ptr = mxGetPr(rhs[id]);
998     param->ALGORITHM = (unsigned int)ptr[0];
999     id++;
1000
1001     // dimension
1002     ptr = mxGetPr(rhs[id]);
1003     param->DIMENSION = (unsigned int)ptr[0];
1004     id++;
1005
1006     // numClasses
1007     ptr = mxGetPr(rhs[id]);
1008     numClasses = (unsigned int)ptr[0];
1009     id++;
1010
1011     // labels
1012     if (mxIsEmpty(rhs[id]) == 0)
1013     {
1014         ptr = mxGetPr(rhs[id]);
1015         for (i = 0; i < numClasses; i++)
1016             (*yLabels).push_back((int)ptr[i]);
1017     }
1018     id++;
1019
1020     // numWeights
1021     ptr = mxGetPr(rhs[id]);
1022     param->BUDGET_SIZE = (unsigned int) ptr[0];
1023     id++;
1024
1025     // bias term
1026     ptr = mxGetPr(rhs[id]);
1027     param->BIAS_TERM = (double) ptr[0];
1028     id++;
1029
1030     // kernel choice
1031     ptr = mxGetPr(rhs[id]);
1032     param->KERNEL = (unsigned int) ptr[0];
1033     id++;
1034
1035     // kernel width gamma
1036     ptr = mxGetPr(rhs[id]);
1037     param->KERNEL_GAMMA_PARAM = (double) ptr[0];
1038     id++;
1039
1040     // kernel degree/slope param
1041     ptr = mxGetPr(rhs[id]);
1042     param->KERNEL_DEGREE_PARAM = (double) ptr[0];
1043     id++;

```

```

1044
1045 // kernel intercept param
1046 ptr = mxGetPr(rhs[id]);
1047 param->KERNEL_COEF_PARAM = (double) ptr[0];
1048 id++;
1049
1050 // weights
1051 unsigned int sr, sc;
1052 mwIndex *ir, *jc;
1053
1054 sr = (int)mxGetN(rhs[id]);
1055 sc = (int)mxGetM(rhs[id]);
1056
1057 ptr = mxGetPr(rhs[id]);
1058 ir = mxGetIr(rhs[id]);
1059 jc = mxGetJc(rhs[id]);
1060
1061 // allocate memory for model
1062 modelLLSVMmatrixW.resize((*param).BUDGET_SIZE, (*param).BUDGET_SIZE);
1063 modelLLSVMweightVector.resize((*param).BUDGET_SIZE, 1);
1064
1065 // weight-vectors are in columns
1066 for (i = 0; i < sr; i++)
1067 {
1068     unsigned int low = (int)jc[i], high = (int)jc[i + 1];
1069     budgetedVectorLLSVM *eNew = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
1070     sqrNorm = 0.0;
1071
1072     // get the linear weight
1073     modelLLSVMweightVector(i, 0) = ptr[low];
1074
1075     // get the modelLLSVMmatrixW
1076     for (j = low + 1; j < low + (*param).BUDGET_SIZE + 1; j++)
1077         modelLLSVMmatrixW(i, j - low - 1) = ptr[j];
1078
1079     // get the features
1080     for (j = low + (*param).BUDGET_SIZE + 1; j < high; j++)
1081     {
1082         ((*eNew)[(int)ir[j] - (*param).BUDGET_SIZE - 1]) = (float)ptr[j];
1083         sqrNorm += (ptr[j] * ptr[j]);
1084     }
1085     eNew->setSqrL2norm(sqrNorm);
1086     (*modelLLSVMlandmarks).push_back(eNew);
1087     eNew = NULL;
1088 }
1089 id++;
1090
1091 mxFree(rhs);
1092 return true;
1093 }

```

5.10.2.2 saveToMatlabStruct()

```

void budgetedModelMatlabLLSVM::saveToMatlabStruct (
    mxArray * plhs[],
    vector< int > * yLabels,
    parameters * param ) [virtual]

```

Save the trained model to Matlab, by creating Matlab structure.

Parameters

out	<i>plhs</i>	Pointer to Matlab output.
in	<i>yLabels</i>	Vector of possible labels.
in	<i>param</i>	The parameters of the algorithm.

The Matlab structure is organized as ["algorithm", "dimension", "numClasses", "labels", "numWeights", "paramBias", "kernelWidth", "model"]. In order to compress memory and to use the memory efficiently, we coded the model in the following way:

The model is stored as ((1 + "dimension") by "numWeights") matrix. Each row corresponds to one landmark point. The first element of each row corresponds to element of linear SVM hyperplane for that particular landmark point. This is followed by features of the landmark point in the original feature space.

Implements [budgetedModelMatlab](#).

Definition at line 829 of file `budgetedSVM_matlab.cpp`.

```

830 {
831     unsigned int i, j, numWeights = 0, cnt;
832     double *ptr;
833     mxArray *returnModel, **rhs;
834     int outID = 0;
835
836     rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * NUM_OF_RETURN_FIELD);
837
838     // algorithm type
839     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
840     ptr = mxGetPr(rhs[outID]);
841     ptr[0] = param->ALGORITHM;
842     outID++;
843
844     // dimension
845     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
846     ptr = mxGetPr(rhs[outID]);
847     ptr[0] = param->DIMENSION;
848     outID++;
849
850     // number of classes
851     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
852     ptr = mxGetPr(rhs[outID]);
853     ptr[0] = (double) (*yLabels).size();
854     outID++;
855
856     // labels
857     rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
858     ptr = mxGetPr(rhs[outID]);
859     for (i = 0; i < (*yLabels).size(); i++)
860         ptr[i] = (*yLabels)[i];
861     outID++;
862
863     // total number of weights
864     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
865     ptr = mxGetPr(rhs[outID]);
866     ptr[0] = (double) (*modelLLSVMlandmarks).size();
867     numWeights = (unsigned int) (*modelLLSVMlandmarks).size();
868     outID++;
869
870     // bias param
871     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
872     ptr = mxGetPr(rhs[outID]);
873     ptr[0] = param->BIAS_TERM;
874     outID++;
875
876     // kernel choice
877     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
878     ptr = mxGetPr(rhs[outID]);
879     ptr[0] = param->KERNEL;
880     outID++;
881
882     // kernel width gamma
883     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
884     ptr = mxGetPr(rhs[outID]);
885     ptr[0] = param->KERNEL_GAMMA_PARAM;
886     outID++;
887
888     // kernel degree/slope param
889     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
890     ptr = mxGetPr(rhs[outID]);
891     ptr[0] = param->KERNEL_DEGREE_PARAM;
892     outID++;
893
894     // kernel intercept param
895     rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
896     ptr = mxGetPr(rhs[outID]);
897     ptr[0] = param->KERNEL_COEF_PARAM;
898     outID++;
899
900     // weights, different for MM algorithms, BSGD and LLSVM
901     int irIndex, nonZeroElement;
902     mwIndex *ir, *jc;
903
904     // find how many non-zero elements there are
905     nonZeroElement = 0;
906     for (i = 0; i < (*modelLLSVMlandmarks).size(); i++)

```

```

907 {
908     // count non-zero features
909     for (j = 0; j < (*param).DIMENSION; j++)
910     {
911         if ((*modelLLSVMlandmarks)[i])[j] != 0.0)
912             nonZeroElement++;
913     }
914
915     // count all elements of modelLLSVMmatrixW also
916     nonZeroElement += (numWeights * numWeights);
917
918     // count linear SVM length also
919     nonZeroElement += numWeights;
920 }
921
922 // +(*yLabels).size() is for the alpha parameters of each BSGD weight
923 rhs[outID] = mxCreateSparse(param->DIMENSION + numWeights + 1, numWeights, nonZeroElement, mxREAL);
924 ir = mxGetIr(rhs[outID]);
925 jc = mxGetJc(rhs[outID]);
926 ptr = mxGetPr(rhs[outID]);
927 jc[0] = irIndex = cnt = 0;
928 for (i = 0; i < (*modelLLSVMlandmarks).size(); i++)
929 {
930     int xIndex = 0;
931
932     // this adds alpha weights to the beginning of a vector, more compact
933     ir[irIndex] = 0;
934     ptr[irIndex] = modelLLSVMweightVector(i, 0);
935     irIndex++, xIndex++;
936
937     // this adds row of modelLLSVMmatrixW next, more compact
938     for (j = 0; j < numWeights; j++)
939     {
940         ir[irIndex] = j + 1; // shift it to accomodate linear weight
941         ptr[irIndex] = modelLLSVMmatrixW(i, j);
942         irIndex++, xIndex++;
943     }
944
945     // add the actual features
946     for (j = 0; j < (*param).DIMENSION; j++)
947     {
948         if ((*modelLLSVMlandmarks)[i])[j] != 0.0)
949         {
950             ir[irIndex] = j + numWeights + 1; // shift it to accomodate linear weight and
modelLLSVMmatrixW
951             ptr[irIndex] = ((*modelLLSVMlandmarks)[i])[j];
952             irIndex++, xIndex++;
953         }
954     }
955     jc[cnt + 1] = jc[cnt] + xIndex;
956     cnt++;
957 }
958
959 /* Create a struct matrix contains NUM_OF_RETURN_FIELD fields */
960 returnModel = mxCreateStructMatrix(1, 1, NUM_OF_RETURN_FIELD, fieldNames);
961
962 /* Fill struct matrix with input arguments */
963 for(i = 0; i < NUM_OF_RETURN_FIELD; i++)
964     mxSetField(returnModel, 0, fieldNames[i], mxDuplicateArray(rhs[i]));
965
966 plhs[0] = returnModel;
967 mxFree(rhs);
968 }

```

The documentation for this class was generated from the following files:

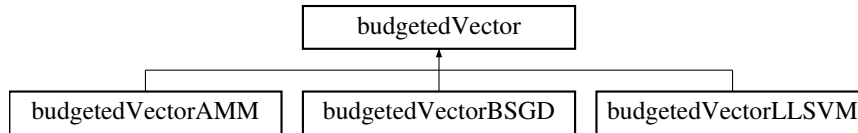
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.cpp

5.11 budgetedVector Class Reference

Class which handles high-dimensional vectors.

```
#include <budgetedSVM.h>
```

Inheritance diagram for budgetedVector:



Public Member Functions

- virtual void [extendDimensionality](#) (unsigned int newDim, [parameters](#) *param)
Extend the dimensionality of the vector.
- virtual long double [getSqrL2norm](#) (void)
Returns [sqrL2norm](#), a squared L2-norm of the vector.
- unsigned int [getDimensionality](#) (void)
Returns [dimension](#), a dimensionality of a vector.
- unsigned int [getID](#) (void)
Returns [weightID](#), a unique ID of a vector.
- const float [operator\[\]](#) (int idx) const
Overloaded [] operator that returns a vector element stored in [array](#).
- float & [operator\[\]](#) (int idx)
Overloaded [] operator that assigns a value to vector element stored in [array](#).
- [budgetedVector](#) (unsigned int dim, unsigned int chnkWght)
Constructor, initializes the vector to all zeros.
- virtual [~budgetedVector](#) ()
Destructor, cleans up the memory.
- virtual void [clear](#) (void)
Clears the vector of all non-zero elements, resulting in a zero-vector.
- virtual void [createVectorUsingDataPoint](#) ([budgetedData](#) *inputData, unsigned int t, [parameters](#) *param)
Create new vector from training data point.
- virtual void [createVectorUsingVector](#) ([budgetedVector](#) *existingVector)
Create new vector from the existing one.
- virtual long double [sqrNorm](#) (void)
Calculates a squared L2-norm of the vector.
- virtual long double [gaussianKernel](#) ([budgetedVector](#) *otherVector, [parameters](#) *param)
Computes Gaussian kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).
- virtual long double [gaussianKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param, long double inputVectorSqrNorm=0.0)
Computes Gaussian kernel between this [budgetedVector](#) vector and another vector from input data stored in [budgetedData](#).
- virtual long double [polyKernel](#) ([budgetedVector](#) *otherVector, [parameters](#) *param)
Computes polynomial kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).
- virtual long double [polyKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param)
Computes polynomial kernel between this [budgetedVector](#) vector and another vector from input data stored in [budgetedData](#).
- virtual long double [linearKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param)
Computes linear kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).
- virtual long double [linearKernel](#) ([budgetedVector](#) *otherVector)
Computes linear kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).
- virtual long double [sigmoidKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param)
Computes sigmoid kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).
- virtual long double [sigmoidKernel](#) ([budgetedVector](#) *otherVector, [parameters](#) *param)

Computes sigmoid kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

- virtual long double [exponentialKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param, long double inputVectorSqrNorm=0.0)

Computes exponential kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

- virtual long double [exponentialKernel](#) ([budgetedVector](#) *otherVector, [parameters](#) *param)

Computes exponential kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

- virtual long double [userDefinedKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param)

Computes user-defined kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

- virtual long double [userDefinedKernel](#) ([budgetedVector](#) *otherVector, [parameters](#) *param)

Computes user-defined kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

- virtual long double [computeKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param, long double inputVectorSqrNorm=0.0)

An umbrella function for all different kernels. Computes kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

- virtual long double [computeKernel](#) ([budgetedVector](#) *otherVector, [parameters](#) *param)

An umbrella function for all different kernels. Computes kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Protected Member Functions

- virtual void [setSqrL2norm](#) (long double newSqrNorm)

Returns [sqrL2norm](#), a squared L2-norm of the vector.

Protected Attributes

- unsigned int [chunkWeight](#)

Length of the vector chunk (implemented as an array).

- unsigned int [dimension](#)

Dimensionality of the vector.

- unsigned int [arrayLength](#)

Number of vector chunks.

- unsigned int [weightID](#)

Unique ID of the vector, used in AMM batch to uniquely identify which vector is assigned to which data points. Assigned when the vector is created.

- vector< float * > [array](#)

Array of vector chunks, element of the array is NULL if all features within a chunk represented by the element are equal to 0.

- long double [sqrL2norm](#)

Squared L2-norm of the vector.

Static Protected Attributes

- static unsigned int [id](#) = 0

ID of the vector.

5.11.1 Detailed Description

Class which handles high-dimensional vectors.

In order to handle high-dimensional vectors (i.e., data points), we split the data vector into an array of smaller vectors (or chunks; implemented as a vector of arrays), and allocate memory for each chunk only if it contains at least one element that is non-zero. This is especially beneficial for very sparse data sets, where we can have considerable memory gains. Each chunk has a pointer to it stored in [array](#), and a pointer is NULL if the chunk has all zero elements; non-NULL pointer points to a chunk that has allocated memory and which stores elements of the vector.

Definition at line 536 of file `budgetedSVM.h`.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 `budgetedVector()`

```
budgetedVector::budgetedVector (
    unsigned int dim,
    unsigned int chnkWght ) [inline]
```

Constructor, initializes the vector to all zeros.

Parameters

in	<i>dim</i>	Dimensionality of the vector.
in	<i>chnkWght</i>	Size of each vector chunk.

Definition at line 651 of file `budgetedSVM.h`.

```
652     {
653         dimension = dim;
654         arrayLength = 0;
655         if (chnkWght > 0)
656             chunkWeight = chnkWght;
657         else
658             svmPrintErrorString("In budgetedVector(), weight chunk must be positive integer!\n");
659
660         if (dim != 0)
661             arrayLength = (unsigned int)((dim - 1) / chunkWeight) + 1;
662
663         // just initialize the elements of array to NULL, will be created only
664         // when needed, when one of the elements becomes non-zero
665         for (unsigned int i = 0; i < arrayLength; i++)
666             array.push_back(NULL);
667
668         weightID = id++;
669         sqrL2norm = 0.0;
670     }
```

5.11.3 Member Function Documentation

5.11.3.1 computeKernel() [1/2]

```
long double budgetedVector::computeKernel (
    budgetedVector * otherVector,
    parameters * param ) [virtual]
```

An umbrella function for all different kernels. Computes kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Parameters

in	<i>otherVector</i>	The second input vector to kernel.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of kernel between two input vectors.

This is an umbrella function for all different kernels. Function computes the value of kernel between two vectors.

Definition at line 1105 of file budgetedSVM.cpp.

```
1106 {
1107     switch ((*param).KERNEL)
1108     {
1109         case KERNEL_FUNC_GAUSSIAN:
1110             return gaussianKernel(otherVector, param);
1111             break;
1112
1113         case KERNEL_FUNC_EXPONENTIAL:
1114             return exponentialKernel(otherVector, param);
1115             break;
1116
1117         case KERNEL_FUNC_SIGMOID:
1118             return sigmoidKernel(otherVector, param);
1119             break;
1120
1121         case KERNEL_FUNC_POLYNOMIAL:
1122             return polyKernel(otherVector, param);
1123             break;
1124
1125         case KERNEL_FUNC_LINEAR:
1126             return linearKernel(otherVector);
1127             break;
1128
1129         case KERNEL_FUNC_USER_DEFINED:
1130             return userDefinedKernel(otherVector, param);
1131             break;
1132
1133         default:
1134             svmPrintErrorString("Error, undefined kernel function found! Run 'budgetedsvm-train' for
1135             help.\n");
1136             return -1.0;
1137     }
1138 }
```

5.11.3.2 computeKernel() [2/2]

```
long double budgetedVector::computeKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param,
    long double inputVectorSqrNorm = 0.0 ) [virtual]
```

An umbrella function for all different kernels. Computes kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>param</i>	The parameters of the algorithm.
in	<i>inputVectorSqrNorm</i>	If zero or not provided, the norm of t-th vector from inputData is computed on-the-fly if necessary (i.e., if RBF kernel is computed).

Returns

Value of kernel between two input vectors.

This is an umbrella function for all different kernels. Function computes the value of kernel between [budgetedVector](#) vector, and the input data point stored in [budgetedData](#).

Definition at line 1149 of file budgetedSVM.cpp.

```

1150 {
1151     switch ((*param).KERNEL)
1152     {
1153         case KERNEL_FUNC_GAUSSIAN:
1154             return gaussianKernel(t, inputData, param, inputVectorSqrNorm);
1155             break;
1156
1157         case KERNEL_FUNC_EXPONENTIAL:
1158             return exponentialKernel(t, inputData, param, inputVectorSqrNorm);
1159             break;
1160
1161         case KERNEL_FUNC_SIGMOID:
1162             return sigmoidKernel(t, inputData, param);
1163             break;
1164
1165         case KERNEL_FUNC_POLYNOMIAL:
1166             return polyKernel(t, inputData, param);
1167             break;
1168
1169         case KERNEL_FUNC_LINEAR:
1170             return linearKernel(t, inputData, param);
1171             break;
1172
1173         case KERNEL_FUNC_USER_DEFINED:
1174             return userDefinedKernel(t, inputData, param);
1175             break;
1176
1177         default:
1178             svmPrintErrorString("Error, undefined kernel function found! Run 'budgetedsvm-train' for
1179             help.\n");
1180             return -1.0;
1181     }
1182 }
```

5.11.3.3 createVectorUsingDataPoint()

```

void budgetedVector::createVectorUsingDataPoint (
    budgetedData * inputData,
    unsigned int t,
    parameters * param ) [virtual]
```

Create new vector from training data point.

Parameters

in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>t</i>	Index of the input vector in the input data.
in	<i>param</i>	The parameters of the algorithm.

Initializes elements of a vector using a data point. Simply copies non-zero elements of the data point stored in [budgetedData](#) to the vector. If the vector already had non-zero elements, it is first cleared to become a zero-vector before copying the elements of a data point.

Definition at line 828 of file `budgetedSVM.cpp`.

```
829 {
830     unsigned int ibegin = inputData->ai[t];
831     unsigned int iend = (t == (unsigned int) (inputData->ai.size() - 1)) ? (unsigned int)
(inputData->aj.size()) : inputData->ai[t + 1];
832
833     this->clear();
834     for (unsigned int i = ibegin; i < iend; i++)
835     {
836         ((*this)[inputData->aj[i] - 1]) = inputData->an[i];
837         sqrL2norm += (inputData->an[i] * inputData->an[i]);
838     }
839
840     if ((*param).BIAS_TERM != 0)
841     {
842         ((*this)[(*param).DIMENSION - 1]) = (float)((long double) (*param).BIAS_TERM);
843         sqrL2norm += ((*param).BIAS_TERM * (*param).BIAS_TERM);
844     }
845 };
```

5.11.3.4 createVectorUsingVector()

```
void budgetedVector::createVectorUsingVector (
    budgetedVector * existingVector ) [virtual]
```

Create new vector from the existing one.

Parameters

in	<i>existingVector</i>	Existing vector which will be cloned into the current one.
----	-----------------------	--

Initializes elements of a vector using an existing vector. If the calling vector already had non-zero elements, it is first cleared to become a zero-vector before duplicating the elements of an input vector.

Definition at line 805 of file `budgetedSVM.cpp`.

```
806 {
807     clear();
808     for (unsigned int i = 0; i < arrayLength; i++)
809     {
810         if (existingVector->array[i] != NULL)
811         {
812             array[i] = new (nothrow) float[chunkWeight];
813             for (unsigned int j = 0; j < chunkWeight; j++)
814                 array[i][j] = existingVector->array[i][j];
815         }
816     }
817     sqrL2norm = existingVector->sqrL2norm;
818 }
```

5.11.3.5 exponentialKernel() [1/2]

```
long double budgetedVector::exponentialKernel (
    budgetedVector * otherVector,
    parameters * param ) [virtual]
```

Computes exponential kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Parameters

in	<i>otherVector</i>	The second input vector to exponential kernel.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of exponential kernel between two input vectors.

Function computes the value of exponential kernel between two vectors. The computation is very fast for sparse data, being only linear in a number of non-zero features. We use the fact that $\|x - y\| = \sqrt{\|x\|^2 - 2 * x^T * y + \|y\|^2}$, where all right-hand side elements can be computed efficiently. For description of the parameters of the kernel see [parameters](#).

Definition at line 908 of file budgetedSVM.cpp.

```

909 {
910     long double temp = sqrt((long double) (sqrL2norm + otherVector->getSqrL2norm() - 2.0L *
        this->linearKernel(otherVector)));
911     if (temp >= 0.0)
912         return exp(-0.5L * (long double) ((*param).KERNEL_GAMMA_PARAM) * temp);
913     else
914         return 0.0L;
915 }
```

5.11.3.6 exponentialKernel() [2/2]

```

long double budgetedVector::exponentialKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param,
    long double inputVectorSqrNorm = 0.0 ) [virtual]
```

Computes exponential kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>param</i>	The parameters of the algorithm.
in	<i>inputVectorSqrNorm</i>	If zero or not provided, the norm of t-th vector from inputData is computed on-the-fly.

Returns

Value of exponential kernel between two input vectors.

Function computes the value of exponential kernel between [budgetedVector](#) vector, and the input data point stored in [budgetedData](#). The computation is very fast for sparse data, being only linear in a number of non-zero features. We use the fact that $\|x - y\| = \sqrt{\|x\|^2 - 2 * x^T * y + \|y\|^2}$, where all right-hand side elements can be computed efficiently. For description of the parameters of the kernel see [parameters](#).

Definition at line 927 of file budgetedSVM.cpp.

```

928 {
929     if (inputVectorSqrNorm == 0.0)
930         inputVectorSqrNorm = inputData->getVectorSqrL2Norm(t, param);
931
932     long double temp = sqrt((long double) (this->sqrL2norm + inputVectorSqrNorm - 2.0L *
this->linearKernel(t, inputData, param)));
933     if (temp >= 0)
934         return exp(-0.5L * (long double) ((*param).KERNEL_GAMMA_PARAM) * temp);
935     else
936         return 0.0L;
937 }

```

5.11.3.7 extendDimensionality()

```

void budgetedVector::extendDimensionality (
    unsigned int newDim,
    parameters * param ) [virtual]

```

Extend the dimensionality of the vector.

Parameters

in	<i>newDim</i>	New dimensionality of the vector.
in	<i>param</i>	The parameters of the algorithm.

Extends the dimensionality of the existing vector to some larger number. We might want to do this due to a variety of reasons, but the introduction of this method was motivated by this situation: it can happen that the user did not correctly specify the number of data dimensions as an input to BudgetedSVM, in which case this parameter is inferred during loading of the data. As in the first version of BudgetedSVM it was mandatory to specify data dimensionality, to remove this restriction we use this function to extend the dimensionality of the existing model vectors to some larger dimensionality. Since the last element of the vector might be a bias term, we also need param object as an input to locate the bias term and move it to a final element of a new, extended vector.

Definition at line 658 of file budgetedSVM.cpp.

```

659 {
660     if (dimension > newDim)
661     {
662         svmPrintErrorString("In extendDimensionality(), extended vector dimensionality smaller than the
old one!\n");
663     }
664     else
665     {
666         /*char text[127];
667         sprintf(text, "In the func, current: %d\tnew: %d!\n", dimension, newDim);
668         svmPrintString(text);*/
669     }
670
671     // when extending the vector, only the last element of the chunk array is modified,
672     // and possibly more zero-chunks are added after the last array element
673     unsigned int newArrayLength = (unsigned int)((newDim - 1) / chunkWeight) + 1;
674
675     float biasTerm = 0.0;
676     if (param->BIAS_TERM != 0.0)
677     {
678         biasTerm = (*this)[dimension - 1];
679     }
680
681     unsigned int lastElementLength = dimension % chunkWeight;
682     if (lastElementLength == 0)
683         lastElementLength = chunkWeight;
684
685     unsigned int newLastElementLength = newDim % chunkWeight;
686     if (newLastElementLength == 0)
687         newLastElementLength = chunkWeight;
688
689     float *temp = NULL;
690     if (newArrayLength == arrayLength)

```

```

691     {
692         // just extend the current last array element by some number of elements, create a new array and
        copy the previous, shorter one to the larger one
693         // if the new and the old array lengths are the same, then possibly the new chunk element is
        also smaller than chunkWeight
694         temp = new float[newLastElementLength];
695         for (unsigned int i = 0; i < newLastElementLength; i++)
696         {
697             if (i < (lastElementLength - (int)(param->BIAS_TERM != 0.0))) // -1 to not copy the bias
        term
698             {
699                 temp[i] = array[arrayLength - 1][i]; // copy the entire last element of chunk-array
700             }
701             else
702             {
703                 temp[i] = 0.0; // set the remaining elements to zero
704             }
705         }
706     }
707     else if (newArrayLength > arrayLength)
708     {
709         // in this case, pad the rest of the current last element with zeros, and new NULL weights will
        be created
710         temp = new float[chunkWeight];
711         for (unsigned int i = 0; i < chunkWeight; i++)
712         {
713             if (i < (lastElementLength - (int)(param->BIAS_TERM != 0.0))) // -1 to not copy the bias
        term
714             {
715                 temp[i] = array[arrayLength - 1][i]; // copy the entire last element of chunk-array
716             }
717             else
718             {
719                 temp[i] = 0.0; // set the remaining elements to zero
720             }
721         }
722         // initialize the additional elements of array to NULL
723         for (unsigned int i = 0; i < newArrayLength - arrayLength; i++)
724             array.push_back(NULL);
725     }
726     else
727     {
728         // just a sanity check
729         svmPrintErrorString("Error in extendDimensionality(): New array length shorter than old one,
        should never happen!");
730     }
731 }
732
733 // put the new, longer chunk instead of the old one
734 delete [] array[arrayLength - 1];
735 array[arrayLength - 1] = temp;
736 temp = NULL;
737
738 // set the static parameters of the budgetedVector class to new values
739 arrayLength = newArrayLength;
740 dimension = newDim;
741
742 // put the bias term to the end if it exists
743 if (param->BIAS_TERM != 0.0)
744 {
745     (*this)[dimension - 1] = biasTerm;
746 }
747 }

```

5.11.3.8 gaussianKernel() [1/2]

```

long double budgetedVector::gaussianKernel (
    budgetedVector * otherVector,
    parameters * param ) [virtual]

```

Computes Gaussian kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Parameters

in	<i>otherVector</i>	The second input vector to RBF kernel.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of RBF kernel between two vectors.

Function computes the value of Gaussian kernel between two vectors. The computation is very fast for sparse data, being only linear in a number of non-zero features. We use the fact that $\|x - y\|^2 = \|x\|^2 - 2 * x^T * y + \|y\|^2$, where all right-hand side elements can be computed efficiently. For description of the parameters of the kernel see [parameters](#).

Definition at line 878 of file budgetedSVM.cpp.

```
879 {
880     return exp(-0.5L * (long double)((*param).KERNEL_GAMMA_PARAM) * (sqrL2norm +
        otherVector->getSqrL2norm() - 2.0L * this->linearKernel(otherVector)));
881 }
```

5.11.3.9 gaussianKernel() [2/2]

```
long double budgetedVector::gaussianKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param,
    long double inputVectorSqrNorm = 0.0 ) [virtual]
```

Computes Gaussian kernel between this [budgetedVector](#) vector and another vector from input data stored in [budgetedData](#).

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>inputVectorSqrNorm</i>	If zero or not provided, the norm of t-th vector from inputData is computed on-the-fly.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of RBF kernel between two vectors.

Function computes the value of Gaussian kernel between [budgetedVector](#) vector, and the input data point stored in [budgetedData](#). The computation is very fast for sparse data, being only linear in a number of non-zero features. We use the fact that $\|x - y\|^2 = \|x\|^2 - 2 * x^T * y + \|y\|^2$, where all right-hand side elements can be computed efficiently. For description of the parameters of the kernel see [parameters](#).

Definition at line 893 of file budgetedSVM.cpp.

```
894 {
895     if (inputVectorSqrNorm == 0.0)
896         inputVectorSqrNorm = inputData->getVectorSqrL2Norm(t, param);
897     return exp(-0.5L * (long double)((*param).KERNEL_GAMMA_PARAM) * (this->sqrL2norm +
        inputVectorSqrNorm - 2.0L * this->linearKernel(t, inputData, param)));
898 }
```

5.11.3.10 `getDimensionality()`

```
unsigned int budgetedVector::getDimensionality (
    void ) [inline]
```

Returns [dimension](#), a dimensionality of a vector.

Returns

Dimensionality of a vector.

Definition at line 618 of file `budgetedSVM.h`.

```
619     {
620         return dimension;
621     }
```

5.11.3.11 `getID()`

```
unsigned int budgetedVector::getID (
    void ) [inline]
```

Returns [weightID](#), a unique ID of a vector.

Returns

Unique ID of a vector.

Definition at line 627 of file `budgetedSVM.h`.

```
628     {
629         return weightID;
630     }
```

5.11.3.12 `getSqrL2norm()`

```
long double budgetedVector::getSqrL2norm (
    void ) [inline], [virtual]
```

Returns [sqrL2norm](#), a squared L2-norm of the vector.

Returns

Squared L2-norm of the vector.

Reimplemented in [budgetedVectorAMM](#).

Definition at line 609 of file `budgetedSVM.h`.

```
610     {
611         return sqrL2norm;
612     }
```

5.11.3.13 `linearKernel()` [1/2]

```
long double budgetedVector::linearKernel (
    budgetedVector * otherVector ) [virtual]
```

Computes linear kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Parameters

in	<i>otherVector</i>	The second input vector to linear kernel.
----	--------------------	---

Returns

Value of linear kernel between two input vectors.

Function computes the value of linear kernel between two vectors. The computation is very fast for sparse data, being only linear in a number of non-zero features.

Definition at line 1042 of file budgetedSVM.cpp.

```

1043 {
1044     long double result = 0.0L;
1045     unsigned long chunkSize = chunkWeight;
1046     for (unsigned int i = 0; i < arrayLength; i++)
1047     {
1048         // if either of them is NULL, meaning all-zeros vector chunk, move on to the next chunk
1049         if ((this->array[i] == NULL) || (otherVector->array[i] == NULL))
1050             continue;
1051
1052         // now we know that i-th vector chunks of both vectors have non-zero elements, go one by one
1053         // and compute linear kernel
1054         if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
1055             chunkSize = dimension % chunkWeight;
1056         for (unsigned int j = 0; j < chunkSize; j++)
1057         {
1058             result += this->array[i][j] * otherVector->array[i][j];
1059         }
1060     }
1061     return result;
1062 }
```

5.11.3.14 linearKernel() [2/2]

```

long double budgetedVector::linearKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param ) [virtual]
```

Computes linear kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of linear kernel between two input vectors.

Function computes the value of linear kernel between [budgetedVector](#) vector, and the input data point stored in [budgetedData](#). The computation is very fast for sparse data, being only linear in a number of non-zero features.

Reimplemented in [budgetedVectorAMM](#).

Definition at line 1002 of file budgetedSVM.cpp.

```

1003 {
1004     long double result = 0.0;
1005     long unsigned int pointIndexPointer = inputData->ai[t];
1006     long unsigned int maxPointIndex = ((unsigned int)(t + 1) == inputData->N) ? inputData->aj.size() :
inputData->aj[t + 1];
1007     char text[256];
1008     unsigned int idx, vectorInd, arrayInd;
1009
1010     for (long unsigned int i = pointIndexPointer; i < maxPointIndex; i++)
1011     {
1012         idx = inputData->aj[i] - 1;
1013         vectorInd = (int) (idx / chunkWeight);
1014         arrayInd = (int) (idx % chunkWeight);
1015
1016         // if the input vector is longer than the budgeted vector, this can happen when the test data
has
1017         // vectors with dimensionality that is longer than previously seen during training, check your
test data!
1018         if (vectorInd >= arrayLength)
1019         {
1020             sprintf(text, "Error, input vector is longer than the budgeted vector, detected dimension
%d in function linearKernel(), check your input data.\n", idx + 1);
1021             svmPrintErrorString(text);
1022         }
1023
1024         // this means that all elements of this chunk are 0
1025         if (array[vectorInd] == NULL)
1026             continue;
1027         else
1028             result += array[vectorInd][arrayInd] * inputData->an[i];
1029     }
1030     if ((*param).BIAS_TERM != 0)
1031         result += (((*this)[(*param).DIMENSION - 1]) * (*param).BIAS_TERM);
1032     return result;
1033 }

```

5.11.3.15 operator[]() [1/2]

```

float & budgetedVector::operator[] (
    int idx )

```

Overloaded [] operator that assigns a value to vector element stored in [array](#).

Parameters

in	idx	Index of vector element that is modified.
----	-----	---

Returns

Value of the modified element of the vector.

Definition at line 754 of file budgetedSVM.cpp.

```

755 {
756     unsigned int vectorInd = (unsigned int)(idx / (int) chunkWeight);
757     unsigned int arrayInd = (unsigned int) (idx % (int) chunkWeight);
758
759     // if the input vector is longer than the budgeted vector, this can happen when the test data has
760     // vectors with dimensionality that is longer than previously seen during training, check your test
data!
761     if (vectorInd >= arrayLength)
762     {
763         svmPrintErrorString("Error, input vector is longer than the budgeted vector in function
budgetedVector::operator[], check your input data.\n");
764     }
765
766     // if all elements were zero, then first create the array and only
767     // then return the reference
768     if (array[vectorInd] == NULL)
769     {

```

```

770     float *tempArray = NULL;
771     unsigned long arraySize = chunkWeight;
772
773     // if the last chunk, then it might be smaller than the rest
774     if (vectorInd == (arrayLength - 1))
775     {
776         arraySize = dimension % chunkWeight;
777         if (arraySize == 0)
778             arraySize = chunkWeight;
779         tempArray = new (nothrow) float[arraySize];
780     }
781     else
782         tempArray = new (nothrow) float[chunkWeight];
783
784     if (tempArray == NULL)
785     {
786         svmPrintErrorString("Memory allocation error (budgetedVector assignment)!");
787     }
788
789     // null the array
790     for (unsigned int j = 0; j < arraySize; j++)
791         *(tempArray + j) = 0;
792
793     array[vectorInd] = tempArray;
794 }
795
796 return *(array[vectorInd] + arrayInd);
797 }

```

5.11.3.16 operator[]() [2/2]

```

const float budgetedVector::operator[] (
    int idx ) const

```

Overloaded [] operator that returns a vector element stored in [array](#).

Parameters

in	<i>idx</i>	Index of vector element that is retrieved.
----	------------	--

Returns

Value of the element of the vector.

Definition at line 639 of file budgetedSVM.cpp.

```

640 {
641     unsigned int vectorInd = (unsigned int) (idx / (int) chunkWeight);
642     unsigned int arrayInd = (unsigned int) (idx % (int) chunkWeight);
643
644     // this means that all elements of this chunk are 0
645     if (array[vectorInd] == NULL)
646         return 0.0;
647     else
648         return *(array[vectorInd] + arrayInd);
649 }

```

5.11.3.17 polyKernel() [1/2]

```

long double budgetedVector::polyKernel (
    budgetedVector * otherVector,
    parameters * param ) [virtual]

```

Computes polynomial kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Parameters

in	<i>otherVector</i>	The second input vector to polynomial kernel.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of polynomial kernel between two vectors.

Function computes the value of polynomial kernel between two vectors. The computation is very fast for sparse data, being only linear in a number of non-zero features. We use the fact that $\|x - y\|^2 = \|x\|^2 - 2 * x^T * y + \|y\|^2$, where all right-hand side elements can be computed efficiently. For description of the parameters of the kernel see [parameters](#).

Definition at line 988 of file budgetedSVM.cpp.

```

989 {
990     return (long double) pow((long double) (param->KERNEL_COEF_PARAM + linearKernel(otherVector)), (long
double) param->KERNEL_DEGREE_PARAM);
991 }
```

5.11.3.18 polyKernel() [2/2]

```

long double budgetedVector::polyKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param ) [virtual]
```

Computes polynomial kernel between this [budgetedVector](#) vector and another vector from input data stored in [budgetedData](#).

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of polynomial kernel between two vectors.

Function computes the value of polynomial kernel between [budgetedVector](#) vector, and the input data point stored in [budgetedData](#). The computation is very fast for sparse data, being only linear in a number of non-zero features. For description of the parameters of the kernel see [parameters](#).

Definition at line 975 of file budgetedSVM.cpp.

```

976 {
977     return (long double) pow((long double) (param->KERNEL_COEF_PARAM + linearKernel(t, inputData,
param)), (long double) param->KERNEL_DEGREE_PARAM);
978 }
```

5.11.3.19 setSqrL2norm()

```
void budgetedVector::setSqrL2norm (
    long double newSqrNorm ) [inline], [protected], [virtual]
```

Returns [sqrL2norm](#), a squared L2-norm of the vector.

Returns

Squared L2-norm of the vector.

Definition at line 590 of file budgetedSVM.h.

```
591     {
592         sqrL2norm = newSqrNorm;
593     }
```

5.11.3.20 sigmoidKernel() [1/2]

```
long double budgetedVector::sigmoidKernel (
    budgetedVector * otherVector,
    parameters * param ) [virtual]
```

Computes sigmoid kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Parameters

in	<i>otherVector</i>	The second input vector to sigmoid kernel.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of sigmoid kernel between two input vectors.

Function computes the value of sigmoid kernel between two vectors. The computation is very fast for sparse data, being only linear in a number of non-zero features. For description of the parameters of the kernel see [parameters](#).

Definition at line 947 of file budgetedSVM.cpp.

```
948 {
949     return (long double) tanh((long double) (param->KERNEL_COEF_PARAM + param->KERNEL_DEGREE_PARAM *
        linearKernel(otherVector)));
950 }
```

5.11.3.21 sigmoidKernel() [2/2]

```
long double budgetedVector::sigmoidKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param ) [virtual]
```

Computes sigmoid kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of sigmoid kernel between two input vectors.

Function computes the value of sigmoid kernel between [budgetedVector](#) vector, and the input data point stored in [budgetedData](#). The computation is very fast for sparse data, being only linear in a number of non-zero features. For description of the parameters of the kernel see [parameters](#).

Definition at line 961 of file budgetedSVM.cpp.

```

962 {
963     return (long double) tanh((long double) (param->KERNEL_COEF_PARAM + param->KERNEL_DEGREE_PARAM *
        linearKernel(t, inputData, param)));
964 }
```

5.11.3.22 sqrNorm()

```

long double budgetedVector::sqrNorm (
    void ) [virtual]
```

Calculates a squared L2-norm of the vector.

Returns

Squared L2-norm of the vector.

Reimplemented in [budgetedVectorAMM](#).

Definition at line 851 of file budgetedSVM.cpp.

```

852 {
853     long double tempSum = 0.0;
854     unsigned long chunkSize = chunkWeight;
855
856     for (unsigned int i = 0; i < arrayLength; i++)
857     {
858         if (array[i] != NULL)
859         {
860             if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
861                 chunkSize = dimension % chunkWeight;
862
863             for (unsigned int j = 0; j < chunkSize; j++)
864                 tempSum += ((long double)array[i][j]) * (long double)array[i][j];
865         }
866     }
867     return tempSum;
868 }
```

5.11.3.23 userDefinedKernel() [1/2]

```

long double budgetedVector::userDefinedKernel (
    budgetedVector * otherVector,
    parameters * param ) [virtual]
```

Computes user-defined kernel between this [budgetedVector](#) vector and another vector stored in [budgetedVector](#).

Parameters

in	<i>otherVector</i>	The second input vector to user-defined kernel.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of user-defined kernel between two input vectors.

Function computes the value of user-defined kernel between two vectors, and before using this function it should be modified by a user. To add your kernel function please open file '[src/budgetedSVM.cpp](#)' and modify two [userDefinedKernel\(\)](#) methods; you can take a look at implementations of other kernel functions for examples.

Definition at line 1089 of file budgetedSVM.cpp.

```

1090 {
1091     // NOTE TO USER: here add your kernel function, be sure to modify BOTH userDefinedKernel() methods;
    after adding your function make sure to comment the below warnings
1092     svmPrintString("\nError, non-implemented user-defined kernel function invoked!\n");
1093     svmPrintErrorString("To add your kernel function please open file 'src/budgetedSVM.cpp' and
    modify\ntwo userDefinedKernel() methods, you can take a look at implementations of\nother kernel
    functions for examples.\n");
1094     return -1.0;
1095 }
```

5.11.3.24 userDefinedKernel() [2/2]

```

long double budgetedVector::userDefinedKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param ) [virtual]
```

Computes user-defined kernel between this [budgetedVector](#) vector and another vector stored in [budgetedData](#).

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of user-defined kernel between two input vectors.

Function computes the value of user-defined kernel between [budgetedVector](#) vector and the input data point stored in [budgetedData](#), and before using this function it should be modified by a user. To add your kernel function please open file '[src/budgetedSVM.cpp](#)' and modify two [userDefinedKernel\(\)](#) methods; you can take a look at implementations of other kernel functions for examples.

Definition at line 1072 of file budgetedSVM.cpp.

```

1073 {
1074     // NOTE TO USER: here add your kernel function, be sure to modify BOTH userDefinedKernel() methods;
    after adding your function make sure to comment the below warnings
1075     svmPrintString("\nError, non-implemented user-defined kernel function!\n");
1076     svmPrintErrorString("To add your kernel function please open file 'src/budgetedSVM.cpp' and
    modify\ntwo userDefinedKernel() methods, you can take a look at implementations of\nother kernel
    functions for examples.\n");
1077     return -1.0;
1078 }
```

5.11.4 Member Data Documentation

5.11.4.1 array

```
vector< float * > budgetedVector::array [protected]
```

Array of vector chunks, element of the array is NULL if all features within a chunk represented by the element are equal to 0.

When the data is sparse, then we do not have to explicitly store every feature as most of them are equal to 0. One option is simply to follow LIBSVM format, and store in two linked lists feature index and the corresponding feature value. However, we found that updating this data structure can become prohibitively slow, as for high-dimensional data the weights can become much less sparse than the original data due to the weight update process, and the insertion of new elements into vector and vector traversal becomes very slow. We address this by storing a vector into structure that is a vector of dynamic arrays, where original, large vector is split into parts (or chunks), and each part is stored in an array within the vector structure. If all elements of the large vector within a chunk are zero, we do not allocate memory for that chunk and [array](#) element for this chunk will be NULL. In our experience, this significantly improves the training and testing time on very high-dimensional sparse data, such as on URL data set with more than 3.2 million features and only 0.004% non-zero values. If [parameters::CHUNK_WEIGHT](#) is set to 1, we obtain the LIBSVM-type representation where each chunk stores only one feature.

See also

[parameters::CHUNK_WEIGHT](#)

Definition at line 583 of file budgetedSVM.h.

5.11.4.2 arrayLength

```
unsigned int budgetedVector::arrayLength [protected]
```

Number of vector chunks.

In order to deal with high-dimensional data, each vector is split into several chunks, and the memory for the chunk is not allocated if all elements of a vector are equal to 0. The static variable [chunkWeight](#) specifies how many of these chunks are used to represent each vector.

See also

[parameters::CHUNK_WEIGHT](#)

Definition at line 581 of file budgetedSVM.h.

5.11.4.3 chunkWeight

```
unsigned int budgetedVector::chunkWeight [protected]
```

Length of the vector chunk (implemented as an array).

See also

[parameters::CHUNK_WEIGHT](#)

Definition at line 579 of file budgetedSVM.h.

5.11.4.4 id

```
static unsigned int budgetedVector::id = 0 [static], [protected]
```

ID of the vector.

Each vector is uniquely identifiable using its ID. This is used in AMM batch algorithm, where weights and data points are matched, and we need to know which weight (represented as [budgetedVector](#)), is assigned to which data point during stochastic gradient descent training.

Definition at line 578 of file budgetedSVM.h.

5.11.4.5 sqrL2norm

```
long double budgetedVector::sqrL2norm [protected]
```

Squared L2-norm of the vector.

After every modification to a [budgetedVector](#) object (e.g., due to an update in Stochastic Gradient Descent (SGD) learning step of AMM or BSGD algorithms), this property is updated to reflect the current squared norm of the vector. This is done to speed up computations of kernel functions, as Gaussian kernel used in BSGD and LLSVM is computed much faster when we know squared norms of two vectors that are inputs to a kernel function. Also, in AMM it is used in pruning phase to find the weights that need to be deleted, as we will prune only weights that have small L2-norm.

Definition at line 584 of file budgetedSVM.h.

5.11.4.6 weightID

```
unsigned int budgetedVector::weightID [protected]
```

Unique ID of the vector, used in AMM batch to uniquely identify which vector is assigned to which data points. Assigned when the vector is created.

See also

[id](#)

Definition at line 582 of file budgetedSVM.h.

The documentation for this class was generated from the following files:

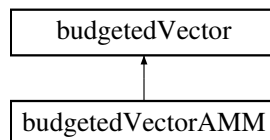
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/[budgetedSVM.h](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.cpp

5.12 budgetedVectorAMM Class Reference

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for AMM algorithms.

```
#include <mm_algs.h>
```

Inheritance diagram for budgetedVectorAMM:



Public Member Functions

- [budgetedVectorAMM](#) (unsigned int dim=0, unsigned int chnkWght=0)
Constructor, initializes the vector to all zeros, and also initializes [degradation](#) parameter.
- long double [getSqrL2norm](#) (void)
Returns [sqrL2norm](#), a squared L2-norm of the vector, which accounts for the vector degradation.
- void [downgrade](#) (long oto)
Downgrade the existing weight-vector.
- long double [sqrNorm](#) (void)
Calculates a squared norm of the vector, but takes into consideration current degradation of a vector.
- long double [getDegradation](#) (void)
Returns [degradation](#) of a vector.
- void [setDegradation](#) (long double deg)
Sets [degradation](#) of a vector.
- void [updateDegradation](#) (unsigned int iteration, [parameters](#) *param)
Computes [degradation](#) of a vector.

- void [updateUsingDataPoint](#) ([budgetedData](#) *inputData, unsigned int oto, unsigned int t, int sign, [parameters](#) *param)
Updates a weight-vector when misclassification happens.
- void [updateUsingVector](#) ([budgetedVectorAMM](#) *otherVector, unsigned int oto, int sign, [parameters](#) *param)
Updates a weight-vector when misclassification happens.
- void [createVectorUsingDataPoint](#) ([budgetedData](#) *inputData, unsigned int oto, unsigned int t, [parameters](#) *param)
Create new weight from one of the zero-weights.
- void [createVectorUsingVector](#) ([budgetedVectorAMM](#) *existingVector)
Create new vector from the existing one.
- long double [linearKernel](#) (unsigned int t, [budgetedData](#) *inputData, [parameters](#) *param)
Computes linear kernel between vector and given input data point, but also accounts for degradation.
- long double [linearKernel](#) ([budgetedVectorAMM](#) *otherVector)
Computes linear kernel between this [budgetedVectorAMM](#) vector and another vector stored in [budgetedVectorAMM](#), but also accounts for degradation.

Protected Attributes

- long double [degradation](#)
Degradation of the vector.

Friends

- class [budgetedModelAMM](#)
- class [budgetedModelMatlabAMM](#)

Additional Inherited Members

5.12.1 Detailed Description

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for AMM algorithms.

Definition at line 26 of file `mm_algs.h`.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 [budgetedVectorAMM\(\)](#)

```
budgetedVectorAMM::budgetedVectorAMM (
    unsigned int dim = 0,
    unsigned int chnkWght = 0 ) [inline]
```

Constructor, initializes the vector to all zeros, and also initializes [degradation](#) parameter.

Parameters

in	<i>dim</i>	Dimensionality of the vector.
in	<i>chnkWght</i>	Size of each vector chunk.

Definition at line 50 of file mm_algs.h.

```

50         chnkWght)                                     : budgetedVector(dim,
51         {
52             degradation = 1.0;
53     }
```

5.12.3 Member Function Documentation**5.12.3.1 createVectorUsingDataPoint()**

```

void budgetedVectorAMM::createVectorUsingDataPoint (
    budgetedData * inputData,
    unsigned int oto,
    unsigned int t,
    parameters * param ) [inline]
```

Create new weight from one of the zero-weights.

Parameters

in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>oto</i>	Total number of iterations so far.
in	<i>t</i>	Index of the input vector in the input data.
in	<i>param</i>	The parameters of the algorithm.

The function simply copies the t-th data point in the input data to the vector vij, while also updating the degradation variable.

Definition at line 147 of file mm_algs.h.

```

148     {
149         budgetedVector::createVectorUsingDataPoint(inputData, t, param);
150         degradation = 1.0 / (((long double)oto + 1.0) * (long double)(*param).LAMBDA_PARAM);
151     }
```

5.12.3.2 createVectorUsingVector()

```

void budgetedVectorAMM::createVectorUsingVector (
    budgetedVectorAMM * existingVector ) [inline]
```

Create new vector from the existing one.

Parameters

in	<i>existingVector</i>	Existing vector which will be cloned into the current one.
----	-----------------------	--

Initializes elements of a vector using an existing vector. If the calling vector already had non-zero elements, it is first cleared to become a zero-vector before duplicating the elements of an input vector.

Definition at line 159 of file mm_algs.h.

```

160     {
161         budgetedVector::createVectorUsingVector(existingVector);
162         setDegradation(existingVector->degradation);
163     }
```

5.12.3.3 downgrade()

```

void budgetedVectorAMM::downgrade (
    long oto ) [inline]
```

Downgrade the existing weight-vector.

Parameters

in	<i>oto</i>	Total number of AMM training iterations so far.
----	------------	---

Using this function, each training iteration all non-zero weights are pushed closer to 0, to ensure the convergence of the algorithm to the optimal solution.

Definition at line 70 of file mm_algs.h.

```

71     {
72         degradation *= (1.0 - 1.0 / ((long double)oto + 1.0));
73     };
```

5.12.3.4 getDegradation()

```

long double budgetedVectorAMM::getDegradation (
    void ) [inline]
```

Returns *degradation* of a vector.

Returns

Degradation of a vector.

Definition at line 88 of file mm_algs.h.

```

89     {
90         return degradation;
91     }
```

5.12.3.5 getSqrL2norm()

```
double budgetedVectorAMM::getSqrL2norm (
    void ) [inline], [virtual]
```

Returns [sqrL2norm](#), a squared L2-norm of the vector, which accounts for the vector degradation.

Returns

Squared L2-norm of the vector.

Reimplemented from [budgetedVector](#).

Definition at line 59 of file `mm_algs.h`.

```
60     {
61         return (degradation * degradation * sqrL2norm);
62     }
```

5.12.3.6 linearKernel() [1/2]

```
long double budgetedVectorAMM::linearKernel (
    budgetedVectorAMM * otherVector ) [inline]
```

Computes linear kernel between this [budgetedVectorAMM](#) vector and another vector stored in [budgetedVectorAMM](#), but also accounts for degradation.

Parameters

<code>in</code>	<code>otherVector</code>	The second input vector to linear kernel.
-----------------	--------------------------	---

Returns

Value of linear kernel between two input vectors.

Function computes the dot product (or linear kernel) between two vectors.

Definition at line 186 of file `mm_algs.h`.

```
187     {
188         return (degradation * otherVector->getDegradation() *
189             budgetedVector::linearKernel(otherVector));
189     };
```

5.12.3.7 linearKernel() [2/2]

```
long double budgetedVectorAMM::linearKernel (
    unsigned int t,
    budgetedData * inputData,
    parameters * param ) [inline], [virtual]
```

Computes linear kernel between vector and given input data point, but also accounts for degradation.

Parameters

in	<i>t</i>	Index of the input vector in the input data.
in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>param</i>	The parameters of the algorithm.

Returns

Value of linear kernel between two input vectors.

Function computes the dot product (i.e., linear kernel) between [budgetedVector](#) vector and the input data point from [budgetedData](#).

Reimplemented from [budgetedVector](#).

Definition at line 174 of file `mm_algs.h`.

```

175     {
176         return (degradation * budgetedVector::linearKernel(t, inputData, param));
177     };

```

5.12.3.8 `sqrNorm()`

```

long double budgetedVectorAMM::sqrNorm (
    void ) [inline], [virtual]

```

Calculates a squared norm of the vector, but takes into consideration current degradation of a vector.

Returns

Squared norm of the vector.

Reimplemented from [budgetedVector](#).

Definition at line 79 of file `mm_algs.h`.

```

80     {
81         return (degradation * degradation * budgetedVector::sqrNorm());
82     }

```

5.12.3.9 `updateDegradation()`

```

void budgetedVectorAMM::updateDegradation (
    unsigned int iteration,
    parameters * param ) [inline]

```

Computes [degradation](#) of a vector.

Parameters

in	<i>iteration</i>	Training iteration at which the degradation is set, used to compute the degradation value.
in	<i>param</i>	The parameters of the algorithm.

Definition at line 106 of file mm_algs.h.

```
107     {
108         degradation = 1.0 / (((long double)iteration + 1.0) * (long double)(*param).LAMBDA_PARAM);
109     }
```

5.12.3.10 updateUsingDataPoint()

```
void budgetedVectorAMM::updateUsingDataPoint (
    budgetedData * inputData,
    unsigned int oto,
    unsigned int t,
    int sign,
    parameters * param )
```

Updates a weight-vector when misclassification happens.

Parameters

in	<i>inputData</i>	Input data from which t-th vector is considered.
in	<i>oto</i>	Total number of iterations so far.
in	<i>t</i>	Index of the input vector in the input data.
in	<i>sign</i>	+1 if the input vector is of the true class, -1 otherwise, specifies how the weights will be updated.
in	<i>param</i>	The parameters of the algorithm.

When we misclassify a data point during training, this function is used to update the existing weight-vector. It brings the true-class weight closer to the misclassified data point, and to push the winning other-class weight away from the misclassified point according to AMM weight-update equations. The misclassified example used to update an existing weight is located in the input data set loaded to [budgetedData](#).

Definition at line 278 of file mm_algs.cpp.

```
279 {
280     unsigned long pointIndexPointer = inputData->ai[t];
281     unsigned long maxPointIndex = ((t + 1) == (unsigned int) inputData->ai.size()) ? (unsigned int)
inputData->aj.size() : inputData->ai[t + 1];
282
283     long double linKern = this->linearKernel(t, inputData, param);
284     long double divisor = (long double)sign * ((long double)oto + 1.0) * (long
double)(*param).LAMBDA_PARAM * degradation;
285     for (unsigned long i = pointIndexPointer; i < maxPointIndex; i++)
286     {
287         ((*this)[inputData->aj[i] - 1]) = (float)((long double)((*this)[inputData->aj[i] - 1]) + (long
double)inputData->an[i] / divisor);
288     }
289     if ((*param).BIAS_TERM != 0)
290     {
291         ((*this)[(*param).DIMENSION - 1]) = (float)((long double)((*this)[(*param).DIMENSION - 1]) +
(long double)(*param).BIAS_TERM / divisor);
292     }
293
294     this->sqrL2norm += (long double)inputData->getVectorSqrL2Norm(t, param) / (divisor * divisor) + 2.0L
/ (divisor * this->degradation) * linKern;
295 }
```

5.12.3.11 updateUsingVector()

```
void budgetedVectorAMM::updateUsingVector (
    budgetedVectorAMM * otherVector,
```

```

    unsigned int oto,
    int sign,
    parameters * param )

```

Updates a weight-vector when misclassification happens.

Parameters

in	<i>otherVector</i>	Misclassified example used to update the existing weight.
in	<i>oto</i>	Total number of iterations so far.
in	<i>sign</i>	+1 if the input vector is of the true class, -1 otherwise, specifies how the weights will be updated.
in	<i>param</i>	The parameters of the algorithm.

When we misclassify a data point during training, this function is used to update the existing weight-vector. It brings the true-class weight closer to the misclassified data point, and to push the winning other-class weight away from the misclassified point according to AMM weight-update equations. The misclassified example used to update an existing weight is located in the [budgetedVectorAMM](#) object.

Definition at line 308 of file `mm_algs.cpp`.

```

309 {
310     unsigned long chunkSize = chunkWeight;
311     unsigned int i, j;
312     float *tempArray = NULL;
313     long double divisor = (long double)sign * ((long double)oto + 1.0) * (long
double) (*param).LAMBDA_PARAM * degradation;
314     long double linKern = this->linearKernel(otherVector);
315     for (i = 0; i < arrayLength; i++)
316     {
317         // if the input vector's i-th array is NULL, then there is no need to update any of this
vector's features
318         if (otherVector->array[i] == NULL)
319             continue;
320
321         // now we know that i-th vector chunk of input vector has non-zero elements, go one by one and
this vector
322         if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
323             chunkSize = dimension % chunkWeight;
324
325         // if the i-th chunk weight is NULL then create it
326         if (this->array[i] == NULL)
327         {
328             // create and null the array
329             tempArray = new (nothrow) float[chunkSize];
330             for (j = 0; j < chunkSize; j++)
331                 *(tempArray + j) = 0;
332             this->array[i] = tempArray;
333         }
334         else
335             tempArray = this->array[i];
336
337         for (j = 0; j < chunkSize; j++)
338         {
339             *(tempArray + j) += (float)((long double) otherVector->array[i][j] / divisor);
340         }
341         tempArray = NULL;
342     }
343
344     sqrtL2norm += (long double)otherVector->getSqrL2norm() / (divisor * divisor) + 2.0L / (divisor *
this->degradation) * linKern;
345 }

```

5.12.4 Member Data Documentation

5.12.4.1 degradation

```
long double budgetedVectorAMM::degradation [protected]
```

Degradation of the vector.

At each iteration during the training procedure of AMM algorithms and Pegasos all weights are degraded, meaning that their elements are pushed slightly towards 0. This can, in addition to numerical issues, also be a problem when the dimensionality of the data set is large, as in naive implementation each feature needs to be degraded independently. However, instead of degrading each element separately, we can keep degradation level as a single number which is the same for all features, thus avoiding round-off problems and also speeding up the degradation step, which now amounts to a single multiplication operation.

Consequently, the actual feature value of a vector is equal to the value stored in [array](#), multiplied by [degradation](#).

Definition at line 42 of file mm_algs.h.

The documentation for this class was generated from the following files:

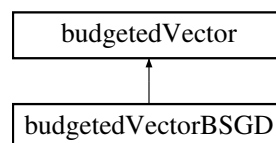
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.cpp

5.13 budgetedVectorBSGD Class Reference

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for BSGD algorithm.

```
#include <bsgd.h>
```

Inheritance diagram for budgetedVectorBSGD:



Public Member Functions

- void [updateSV](#) ([budgetedVectorBSGD](#) *v, long double kMax)
Updates the vector to obtain a merged vector, used during merging budget maintenance.
- [budgetedVectorBSGD](#) (unsigned int dim=0, unsigned int chnkWght=0, unsigned int numCls=0)
Constructor, initializes the vector to all zeros, and also initializes class-specific alpha parameters.
- long double [alphaNorm](#) (void)
Computes the norm of alpha vector.
- void [downgrade](#) (unsigned long oto)
Downgrade the alpha-parameters.

Static Public Member Functions

- static unsigned int [getNumClasses](#) (void)
Get the number of classes in the classification problem.

Public Attributes

- vector< long double > [alphas](#)
Array of class-specific alpha parameters, used in BSGD algorithm.

Static Protected Attributes

- static unsigned int [numClasses](#) = 0
Number of classes of the classification problem, specifies the size of [alphas](#) vector.

Friends

- class **budgetedModelBSGD**
- class **budgetedModelMatlabBSGD**

Additional Inherited Members

5.13.1 Detailed Description

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for BSGD algorithm.

Definition at line 26 of file bsgd.h.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 budgetedVectorBSGD()

```
budgetedVectorBSGD::budgetedVectorBSGD (
    unsigned int dim = 0,
    unsigned int chnkWght = 0,
    unsigned int numCls = 0 ) [inline]
```

Constructor, initializes the vector to all zeros, and also initializes class-specific alpha parameters.

Parameters

in	<i>dim</i>	Dimensionality of the vector.
in	<i>chnkWght</i>	Size of each vector chunk.
in	<i>numCls</i>	Number of classes in the classification problem, specifies the size of alphas vector.

Definition at line 71 of file bsgd.h.

```

71                                     :
72     budgetedVector(dim, chnkWght)
73     {
74         if (numClasses == 0)
75             numClasses = numCls;
76         for (unsigned int i = 0; i < numClasses; i++)
77             this->alphas.push_back(0.0);
78     }

```

5.13.3 Member Function Documentation

5.13.3.1 alphaNorm()

```

long double budgetedVectorBSGD::alphaNorm (
    void )

```

Computes the norm of alpha vector.

Returns

Norm of the alpha vector.

Computes the l2-norm of the alpha vector.

See also

`budgetedVector::alphas`

Definition at line 346 of file bsgd.cpp.

```

347 {
348     long double tempSum = 0.0;
349     for (unsigned long i = 0; i < alphas.size(); i++)
350         tempSum += (alphas[i] * alphas[i]);
351     return tempSum;
352 }

```

5.13.3.2 downgrade()

```

void budgetedVectorBSGD::downgrade (
    unsigned long oto ) [inline]

```

Downgrade the alpha-parameters.

Parameters

in	oto	Total number of iterations so far.
----	-----	------------------------------------

Each training iteration the alpha parameters are pushed towards 0 to ensure the convergence of the algorithm to

the optimal solution.

Definition at line 94 of file bsgd.h.

```

95     {
96         for (unsigned int i = 0; i < alphas.size(); i++)
97             if (alphas[i] != 0)
98                 alphas[i] *= (1.0 - 1.0 / (long double) oto);
99     };

```

5.13.3.3 getNumClasses()

```

unsigned int budgetedVectorBSGD::getNumClasses (
    void ) [inline], [static]

```

Get the number of classes in the classification problem.

Returns

Number of classes that are covered by this vector, also the length of [alphas](#).

Definition at line 50 of file bsgd.h.

```

51     {
52         return numClasses;
53     }

```

5.13.3.4 updateSV()

```

void budgetedVectorBSGD::updateSV (
    budgetedVectorBSGD * v,
    long double kMax )

```

Updates the vector to obtain a merged vector, used during merging budget maintenance.

Parameters

in	<i>v</i>	Vector that is merged with this vector.
in	<i>kMax</i>	Parameter that specifies how to combine them ($\text{currentVector} <- kMax * \text{currentVector} + (1 - kMax) * v$).

When we find which two support vectors to merge, together with the value of the merging parameter *kMax*, this function updates one of the two vectors to obtain the merged support vector. After the merging, the other vector is no longer needed and can be deleted.

See also

[computeKmax](#)

Definition at line 290 of file bsgd.cpp.

```

291 {
292     unsigned long chunkSize = chunkWeight;

```

```

293     unsigned long i, j;
294     long double linKern = this->linearKernel(v);
295
296     for (i = 0; i < arrayLength; i++)
297     {
298         if (this->array[i] != NULL)
299         {
300             if ((*v).array[i] == NULL)
301             {
302                 if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
303                     chunkSize = dimension % chunkWeight;
304                 for (j = 0; j < chunkSize; j++)
305                     array[i][j] = (float)(kMax * (long double) this->array[i][j]);
306             }
307             else
308             {
309                 if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
310                     chunkSize = dimension % chunkWeight;
311                 for (j = 0; j < chunkSize; j++)
312                     this->array[i][j] = (float)(kMax * this->array[i][j] + (1.0 - kMax) *
313 (*v).array[i][j]);
314             }
315             else
316             {
317                 if ((*v).array[i] != NULL)
318                 {
319                     if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
320                         chunkSize = dimension % chunkWeight;
321
322                     float *tempArray = new (nothrow) float[chunkSize];
323                     if (tempArray == NULL)
324                     {
325                         svmPrintErrorString("Memory allocation error (budgetedVector assignment)!");
326                     }
327
328                     // copy the array
329                     for (j = 0; j < chunkSize; j++)
330                         *(tempArray + j) = (float)((1.0 - kMax) * (*v).array[i][j]);
331
332                     this->array[i] = tempArray;
333                     tempArray = NULL;
334                 }
335             }
336         }
337
338         // we also update the squared norm of the merged vector
339         this->sqrL2norm = kMax * kMax * (long double) (this->sqrL2norm) + (1.0L - kMax) * (1.0L - kMax) *
340 v->sqrNorm() + 2.0L * kMax * (1.0L - kMax) * linKern;

```

5.13.4 Member Data Documentation

5.13.4.1 alphas

```
vector< double > budgetedVectorBSGD::alphas
```

Array of class-specific alpha parameters, used in BSGD algorithm.

This vector is of the size that equals number of classes in the data set. Each element specifies the influence a [budgetedVector](#) has on a specific class.

Definition at line 44 of file bsgd.h.

The documentation for this class was generated from the following files:

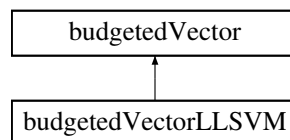
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.cpp

5.14 budgetedVectorLLSVM Class Reference

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for LLSVM algorithm.

```
#include <llsvm.h>
```

Inheritance diagram for budgetedVectorLLSVM:



Public Member Functions

- void [createVectorUsingDataPointMatrix](#) (VectorXd &dataVector)
Initialize the vector using a data point represented as a (1 x DIMENSION) matrix.
- [budgetedVectorLLSVM](#) (unsigned int dim=0, unsigned int chnkWght=0)
Constructor, initializes the LLSVM vector to zero weights.

Friends

- class **budgetedModelLLSVM**
- class **budgetedModelMatlabLLSVM**

Additional Inherited Members

5.14.1 Detailed Description

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for LLSVM algorithm.

Definition at line 26 of file llsvm.h.

5.14.2 Member Function Documentation

5.14.2.1 createVectorUsingDataPointMatrix()

```
void budgetedVectorLLSVM::createVectorUsingDataPointMatrix (
    VectorXd & dataVector ) [inline]
```

Initialize the vector using a data point represented as a (1 x DIMENSION) matrix.

Parameters

in	<i>dataVector</i>	Row vector holding a data point.
----	-------------------	----------------------------------

Used during the initialization stage of the LLSVM algorithm to store the found landmark point in an instance of [budgetedVectorLLSVM](#) class.

Definition at line 39 of file `llsvm.h`.

```

40     {
41
42
43
44         for (unsigned int i = 0; i < (unsigned int) dataVector.size(); i++)
45         {
46             if (dataVector[i] != 0.0)
47             {
48
49
50                 (*this)[i] = (float) dataVector[i];
51                 sqrL2norm += (dataVector[i] * dataVector[i]);
52             }
53         }
54     };

```

The documentation for this class was generated from the following file:

- `C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/llsvm.h`

5.15 parameters Struct Reference

Structure holds the parameters of the implemented algorithms.

```
#include <budgetedSVM.h>
```

Public Member Functions

- [parameters](#) (void)
Constructor of the structure. The default values of the parameters can be modified here manually.
- void [updateVerySparseDataParameter](#) (double dataSparsity)
If [VERY_SPARSE_DATA](#) parameter was not set by a user, this function sets this parameter according to the sparsity of the loaded data.

Public Attributes

- unsigned int [ALGORITHM](#)
Algorithm that is used, 0 - Pegasos; 1 - AMM batch; 2 - AMM online; 3 - LLSVM; 4 - BSGD (default: 2)
- unsigned int [NUM_SUBEPOCHS](#)
Number of training subepochs of AMM batch algorithm (default: 1)
- unsigned int [NUM_EPOCHS](#)
Number of training epochs (default: 5)
- unsigned int [K_PARAM](#)
*Frequency *k* of weight pruning of AMM algorithms (default: 10,000 iterations)*
- unsigned int [DIMENSION](#)
Dimensionality of the classification problem, MUST be set by a user (default: 0)

- unsigned int [CHUNK_SIZE](#)
Size of the chunk of the data loaded at once (default: 50,000 data points)
- unsigned int [CHUNK_WEIGHT](#)
Size of chunk of [budgetedVector](#) weight (whole vector is split into smaller parts) (default: 1,000)
- unsigned int [KERNEL](#)
Choose the kernel function for kernel-based algorithms, 0 - Gaussian kernel, 1 - polynomial kernel, 2 - linear kernel (default: 0)
- unsigned int [BUDGET_SIZE](#)
Maximum number of weight per class of AMM algorithms, OR size of the budget of BSGD algorithm, OR number of landmark points in LLSVM algorithm (default: 50)
- unsigned int [K_MEANS_ITERS](#)
Number of k-means iterations in initialization of LLSVM algorithm (default: 10)
- unsigned int [MAINTENANCE_SAMPLING_STRATEGY](#)
Budget maintenance strategy of BSGD algorithm, 0 - random removal; 1 - merging, OR type of landmark points sampling in LLSVM algorithm, 0 - random; 1 - k-means; 2 - k-medoids (default: 0)
- unsigned int [VERY_SPARSE_DATA](#)
User set parameter, if a user believes the data is very sparse this parameters can be set to 0/1, where 1 - very sparse data; 0 - not very sparse data (default: see long description)
- double [C_PARAM](#)
Weight pruning parameter c of AMM algorithms (default: 10.0)
- double [BIAS_TERM](#)
Bias term of AMM batch, AMM online, and PEGASOS algorithms (default: 1.0)
- double [KERNEL_GAMMA_PARAM](#)
Kernel width parameter in Gaussian kernel $\exp(-0.5 * \text{KERNEL_GAMMA_PARAM} * ||x - y||^2)$ (default: $1/\text{DIMENSIONALITY}$)
- double [KERNEL_DEGREE_PARAM](#)
Degree of polynomial kernel $(x^T * y + \text{KERNEL_COEF_PARAM})^{\text{KERNEL_DEGREE_PARAM}}$, OR slope parameter of sigmoid kernel $\tanh(\text{KERNEL_DEGREE_PARAM} * x^T * y + \text{KERNEL_COEF_PARAM})$ (default: 2)
- double [KERNEL_COEF_PARAM](#)
Coefficient of polynomial kernel $(x^T * y + \text{KERNEL_COEF_PARAM})^{\text{KERNEL_DEGREE_PARAM}}$, or intercept of sigmoid kernel $\tanh(\text{KERNEL_DEGREE_PARAM} * x^T * y + \text{KERNEL_COEF_PARAM})$ (default: 1)
- double [LAMBDA_PARAM](#)
Lambda regularization parameter; higher values result in more regularization (default: 0.0001)
- double [CLONE_PROBABILITY](#)
Probability of cloning a true-class weight when a misclassification happens (default: 0.0)
- double [CLONE_PROBABILITY_DECAY](#)
Value between 0 and 1 by which [CLONE_PROBABILITY](#) is decayed after successful weight duplication (default: 0.99)
- bool [VERBOSE](#)
Print verbose output during algorithm execution, 1 - verbose output; 0 - quiet (default: 0)
- bool [RANDOMIZE](#)
Randomize (i.e., shuffle) the training data, 1 - randomization on; 0 - randomization off (default: 1)
- bool [OUTPUT_SCORES](#)
Output the winning class scores, in addition to class predictions, 1 - output class scores; 0 - output without class scores (default: 0)

5.15.1 Detailed Description

Structure holds the parameters of the implemented algorithms.

Structure holds the parameters of the implemented algorithms. If needed, the default parameters for each algorithm can be manually modified here.

Definition at line 109 of file `budgetedSVM.h`.

5.15.2 Member Function Documentation

5.15.2.1 updateVerySparseDataParameter()

```
void parameters::updateVerySparseDataParameter (
    double dataSparsity ) [inline]
```

If `VERY_SPARSE_DATA` parameter was not set by a user, this function sets this parameter according to the sparsity of the loaded data.

Parameters

in	<code>dataSparsity</code>	The sparsity of the loaded data set.
----	---------------------------	--------------------------------------

When computing the kernels between support vectors/hyperplanes kept in the available budget in `budgetedVector` objects on one side, and the incoming data points on the other, we have two options: (1) we can either do the computations directly between the support vectors and data points that are stored in `budgetedData`; or (2) we can do the computations between the support vectors and data points that are in the intermediate step stored in the `budgetedVector` object. When the data is very sparse option (1) is faster, as there is very small number of non-zero features that affects the speed of the computations, and the overhead of creating the `budgetedVector` instance might prove too costly. On the other hand, when the data is not too sparse, then it might prove faster to first create `budgetedVector` that will hold the incoming data point, and only then do the kernel computations. The reason is partly in a slow modulus operation that is used in the case (1) (please refer to the implementation of linear and Gaussian kernels to see how it was coded).

See also

`VERY_SPARSE_DATA`, `budgetedVector::linearKernel(unsigned int, budgetedData*, parameters*)`, `budgetedVector::linearKernel(budgetedVector::gaussianKernel(unsigned int, budgetedData*, parameters*, long double), budgetedVector::gaussianKernel(bu`

Definition at line 314 of file `budgetedSVM.h`.

```
315 {
316     // if the parameter is already set then just return and change nothing; it can be that it was
    set by a user
317     // or it was already set when the earlier data chunks were loaded
318     if ((VERY_SPARSE_DATA == 0) || (VERY_SPARSE_DATA == 1))
319         return;
320
321     // if the sparsity is less than 5%, then we say that we are working with very sparse data
322     if (dataSparsity < 5.0)
323         VERY_SPARSE_DATA = 1;
324     else
325         VERY_SPARSE_DATA = 0;
326 };
```

5.15.3 Member Data Documentation

5.15.3.1 BIAS_TERM

```
double parameters::BIAS_TERM
```

Bias term of AMM batch, AMM online, and PEGASOS algorithms (default: 1.0)

If the parameter is non-zero, a bias, or intercept term, is added to the data set as an additional feature. The value of this additional feature is equal to BIAS_TERM.

Definition at line 263 of file budgetedSVM.h.

5.15.3.2 BUDGET_SIZE

```
unsigned int parameters::BUDGET_SIZE
```

Maximum number of weight per class of AMM algorithms, OR size of the budget of BSGD algorithm, OR number of landmark points in LLSVM algorithm (default: 50)

- AMM: As the number of weights in AMM algorithms is infinite, we can set the limit on the number of non-zero weights that can be stored in memory. This can be done in order to avoid memory-related problems. Once the limit is reached, we do not allow creation of new non-zero weights until some get pruned.
- BSGD: Maximum number of support vectors that can be stored. After the budget is exceeded, [MAINTENANCE_SAMPLING_STRATEGY](#) specifies how the number of support vectors is kept limited.
- LLSVM: In addition, it also specifies the number of landmark points in LLSVM algorithm, that are used to represent the data set in lower-dimensional space using the Nystrom method.

Definition at line 262 of file budgetedSVM.h.

5.15.3.3 C_PARAM

```
double parameters::C_PARAM
```

Weight pruning parameter c of AMM algorithms (default: 10.0)

In order to reduce the complexity of the learned model, which directly improves generalization of the model as shown in the original AMM paper, pruning of small non-zero weights is performed. C_PARAM specifies the aggressiveness of weight pruning, where larger value results in pruning of more weights. More specifically, we sort the weights by their L2-norms, and then prune from the smallest toward larger weight until the cumulative weight norm exceeds value of C_PARAM. Frequency of pruning is controlled by [K_PARAM](#) parameter.

Definition at line 263 of file budgetedSVM.h.

5.15.3.4 CHUNK_SIZE

```
unsigned int parameters::CHUNK_SIZE
```

Size of the chunk of the data loaded at once (default: 50,000 data points)

While [CHUNK_WEIGHT](#) helps when one is working with high-dimensional data, this parameter helps when working with large data with many instances. If the data set is very large and can not fit into memory, we can then load only a small part of it (called *data chunk*), that is processed before being discarded to make room for the next chunk. Therefore, we load only a smaller part of the large data set, with size of this chunk specified by this parameter.

Definition at line 261 of file `budgetedSVM.h`.

5.15.3.5 CHUNK_WEIGHT

```
unsigned int parameters::CHUNK_WEIGHT
```

Size of chunk of [budgetedVector](#) weight (whole vector is split into smaller parts) (default: 1,000)

While [CHUNK_SIZE](#) helps when one is working with large data with many data points, this parameter helps when working with high-dimensional data. When the data is sparse, then we do not have to explicitly store every feature as most of them are equal to 0. One option is simply to follow LIBSVM format, and store a vector in two linked lists, one holding feature index and the other holding the corresponding feature value. However, we found that accessing this data structure can become prohibitively slow, as for high-dimensional data weights can become less sparse than the original data due to the weight update process. For example, when we want to update a specific feature during gradient descent training we would like to do it very quickly, most preferably we would like to have random access to the element of the weight vector that will be updated. We address this by storing a vector into linked list, where each element of the linked list, called *weight chunk*, holding a subset of features. For example, the first chunk would hold features indexed from 1 to `CHUNK_SIZE`, the second would hold features indexed from `CHUNK_SIZE+1` to `2*CHUNK_SIZE`, and so on. If all elements of a weight chunk are zero, we do not allocate memory for that array. In our experience, this significantly improved the training and testing time on truly high-dimensional data, such as on URL data set with more than 3.2 million features. If [CHUNK_WEIGHT](#) is equal to 1, we obtain the LIBSVM-type representation.

Definition at line 261 of file `budgetedSVM.h`.

5.15.3.6 CLONE_PROBABILITY

```
bool parameters::CLONE_PROBABILITY
```

Probability of cloning a true-class weight when a misclassification happens (default: 0.0)

When a misclassification occurs both the true-class and the incorrect-class weights are updated. However, there is also an option to duplicate the true-class weight before the update step, leading to better performance on highly-nonlinear problems. This is done by throwing a biased coin with this probability and generating a duplicate weight if the throw is successful. Note however that this probability is decreased every time a weight is successfully duplicated, controlled by the parameter [CLONE_PROBABILITY_DECAY](#).

Definition at line 263 of file `budgetedSVM.h`.

5.15.3.7 DIMENSION

```
unsigned int parameters::DIMENSION
```

Dimensionality of the classification problem, MUST be set by a user (default: 0)

Although the dimensionality of the data set can be found from the training data set during loading, we ask a user to specify it beforehand, as it is usually a known parameter. The reason why we require this as an input is to speed up processing of the data, since the emphasis of the software is on speeding up the training of classification algorithm on large data, and this little piece of information can help avoid unnecessary bookkeeping tasks. More specifically, the parameter is important for memory management of [budgetedVector](#), where it is used to find how many weight chunks of size [CHUNK_WEIGHT](#) are needed to represent the data.

However, in the case of Matlab interface, it is not required to manually set this parameter as it is easily found by reading the dimensions of the Matlab structure holding the data set.

Definition at line 261 of file budgetedSVM.h.

5.15.3.8 K_MEANS_ITERS

```
unsigned int parameters::K_MEANS_ITERS
```

Number of k-means iterations in initialization of LLSVM algorithm (default: 10)

In order to find better lower-dimensional representation of the data set using Nystrom method, k-means can be used to improve the choice of landmark points. Unlike in random sampling of landmark points from the data set, cluster centers of k-means will represent [BUDGET_SIZE](#) points used for the Nystrom method.

Definition at line 262 of file budgetedSVM.h.

5.15.3.9 K_PARAM

```
unsigned int parameters::K_PARAM
```

Frequency k of weight pruning of AMM algorithms (default: 10,000 iterations)

In order to reduce the complexity of the learned model, which directly improves generalization of the model as shown in the AMM paper, pruning of small non-zero weights is performed. [K_PARAM](#) specifies the frequency of weight pruning, i.e., after how many iterations we perform the pruning step. Aggressiveness of pruning is controlled by [C_PARAM](#) parameter.

Definition at line 261 of file budgetedSVM.h.

5.15.3.10 KERNEL

```
unsigned int parameters::KERNEL
```

Choose the kernel function for kernel-based algorithms, 0 - Gaussian kernel, 1 - polynomial kernel, 2 - linear kernel (default: 0)

The parameter indicates which kernel function is used in kernel-based algorithms. Note that there is no such choice for AMM. The following kernels are available for two input data points x and y :

- Gaussian: $K(x, y) = \exp(-0.5 * \text{KERNEL_GAMMA_PARAM} * ||x - y||^2)$
- Exponential: $K(x, y) = \exp(-0.5 * \text{KERNEL_GAMMA_PARAM} * ||x - y||)$
- Polynomial: $K(x, y) = (x^T * y + \text{KERNEL_COEF_PARAM})^{\text{KERNEL_DEGREE_PARAM}}$
- Linear: $K(x, y) = (x^T * y)$
- Sigmoid: $K(x, y) = \tanh(\text{KERNEL_DEGREE_PARAM} * x^T * y + \text{KERNEL_COEF_PARAM})$
- User-defined: To add your kernel function please open file 'src/budgetedSVM.cpp' and modify two user-defined methods located there.

Definition at line 261 of file budgetedSVM.h.

5.15.3.11 LAMBDA_PARAM

```
double parameters::LAMBDA_PARAM
```

Lambda regularization parameter; higher values result in more regularization (default: 0.0001)

The parameter defines the level of model regularization, where larger values result in less complex model (i.e., more regularized model). The parameter is used in all BudgetedSVM algorithms with the same effect, and decreasing the value of this parameter leads to more overfitting on the training set. When compared to C parameter used in LibLinear solver which is employed in LLSVM algorithm, LAMBDA_PARAM is exactly reciprocal (i.e., $\text{LAMBDA_PARAM} = 1 / C$).

Definition at line 263 of file budgetedSVM.h.

5.15.3.12 MAINTENANCE_SAMPLING_STRATEGY

```
unsigned int parameters::MAINTENANCE_SAMPLING_STRATEGY
```

Budget maintenance strategy of BSGD algorithm, 0 - random removal; 1 - merging, OR type of landmark points sampling in LLSVM algorithm, 0 - random; 1 - k-means; 2 - k-medoids (default: 0)

- BSGD: Whenever a number of support vectors in BSGD algorithm exceeds [BUDGET_SIZE](#), one of the following budget maintenance steps is performed, depending on the value of the MAINTENANCE_SAMPLING_STRATEGY parameter
 - 0 - deleting random support vector to maintain the budget
 - 1 - take two support vectors and merging them into one. The new, merged support vector is located on the straight line connecting the two existing support vectors; where exactly on the line is explained in *computeKmax()* function from *bsdg.cpp* file. Then, the two existing support vectors are deleted and the merged vector is inserted in the budget. Note that kernel function for BSGD when merging strategy is chosen defaults to Gaussian kernel. (default setting)
- LLSVM: Specifies how the landmark points, used to represent the data set in lower-dimensional space using the Nystrom method, are chosen.
 - 0 - landmark points are randomly sampled from the the first loaded data chunk
 - 1 - landmark points will be cluster centers after running k-means on the first loaded data chunk (default setting)
 - 2 - landmark points will be cluster medoids after running k-medoids on the first loaded data chunk

Definition at line 262 of file budgetedSVM.h.

5.15.3.13 NUM_EPOCHS

```
unsigned int parameters::NUM_EPOCHS
```

Number of training epochs (default: 5)

Number of times the data set is seen by the training procedure, each time randomly reshuffled.

Definition at line 261 of file budgetedSVM.h.

5.15.3.14 NUM_SUBEPOCHS

```
unsigned int parameters::NUM_SUBEPOCHS
```

Number of training subepochs of AMM batch algorithm (default: 1)

AMM batch has an option to reassign data points to weights several times during one epoch. In the most extreme case, if [NUM_SUBEPOCHS](#) is equal to the size of the data set, we obtain AMM online algorithm. This parameter specifies how many times we reassign points to weights within a single epoch.

Definition at line 261 of file budgetedSVM.h.

5.15.3.15 OUTPUT_SCORES

```
bool parameters::OUTPUT_SCORES
```

Output the winning class scores, in addition to class predictions, 1 - output class scores; 0 - output without class scores (default: 0)

If this parameter is set, the output scores should be interpreted as follows. For LLSVM the score represents the distance of test example from the separating hyperplane; for AMM and BSGD this score represents difference between the winning-class score and the score of a class that had the second-best score.

Definition at line 264 of file `budgetedSVM.h`.

5.15.3.16 VERY_SPARSE_DATA

```
unsigned int parameters::VERY_SPARSE_DATA
```

User set parameter, if a user believes the data is very sparse this parameters can be set to 0/1, where 1 - very sparse data; 0 - not very sparse data (default: see long description)

When computing the kernels between support vectors/hyperplanes kept in the available budget in [budgetedVector](#) objects on one side, and the incoming data points on the other, we have two options: (1) we can either do the computations directly between the support vectors and data points that are stored in [budgetedData](#); or (2) we can do the computations between the support vectors and data points that are in the intermediate step stored in the [budgetedVector](#) object. When the data is very sparse option (1) is faster, as there is very small number of non-zero features that affects the speed of the computations, and the overhead of creating the [budgetedVector](#) instance might prove too costly. On the other hand, when the data is not too sparse, then it might prove faster to first create [budgetedVector](#) that will hold the incoming data point, and only then do the kernel computations. The reason is partly in a slow modulus operation that is used in the case (1) (please refer to the implementation of linear and Gaussian kernels to see how it was coded).

If a user does not manually set this parameter to 0 (i.e., instructs the toolbox to compute kernels as in case (1)) or 1 (i.e., compute kernels as in case (2)), the default setting will be 0 if the sparsity of the loaded data is less than 5% (i.e., less than 5% of the features are non-zero on average), otherwise it will default to 1. For this default behavior that is adaptive to the found data sparsity a developer can set this parameter to anything other than 0 or 1. For more details, please see the train and test functions of the implemented algorithms, and look for code parts where `VERY_SPARSE_DATA` appears.

See also

```
updateVerySparseDataParameter(), budgetedVector::linearKernel(unsigned int, budgetedData*, parameters*),
budgetedVector::linearKernel(budgetedVector*), budgetedVector::gaussianKernel(unsigned int, budgetedData*, parameters*),
budgetedVector::gaussianKernel(budgetedVector*, parameters*)
```

Definition at line 262 of file `budgetedSVM.h`.

The documentation for this struct was generated from the following file:

- `C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.h`

Chapter 6

File Documentation

6.1 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/M/matlab/budgetedSVM_matlab.h File Reference

Implements classes and functions used for training and testing of budgetedSVM algorithms in Matlab.

Classes

- class [budgetedDataMatlab](#)
Class which manipulates sparse array of vectors (similarly to Matlab sparse matrix structure), with added functionality to load data directly from Matlab.
- class [budgetedModelMatlab](#)
Interface which defines methods to load model from and save model to Matlab environment.
- class [budgetedModelMatlabAMM](#)
Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to Matlab environment.
- class [budgetedModelMatlabBSGD](#)
Class which holds the BSGD model, and implements methods to load BSGD model from and save BSGD model to Matlab environment.
- class [budgetedModelMatlabLLSVM](#)
Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to Matlab environment.

Functions

- void [printStringMatlab](#) (const char *s)
Prints string to Matlab, used to modify callback in [budgetedSVM.cpp](#).
- void [printErrorStringMatlab](#) (const char *s)
Prints error string to Matlab, used to modify callback found in [budgetedSVM.cpp](#).
- void [fakeAnswer](#) (mxArray *plhs[])
Returns empty matrix to Matlab.
- void [printUsageMatlab](#) (bool trainingPhase, [parameters](#) *param)
Prints to standard output the instructions on how to use the software.
- void [parseInputMatlab](#) ([parameters](#) *param, const char *paramString, bool trainingPhase, const char *inputFileName=NULL, const char *modelName=NULL)
Parses the user input and modifies parameter settings as necessary.

6.1.1 Detailed Description

Implements classes and functions used for training and testing of budgetedSVM algorithms in Matlab.

6.1.2 Function Documentation

6.1.2.1 fakeAnswer()

```
void fakeAnswer (
    mxArray * plhs[ ] )
```

Returns empty matrix to Matlab.

Parameters

out	<i>plhs</i>	Pointer to Matlab output.
-----	-------------	---------------------------

Definition at line 1118 of file budgetedSVM_matlab.cpp.

```
1119 {
1120     plhs[0] = mxCreateDoubleMatrix(0, 0, mxREAL);
1121 }
```

6.1.2.2 parseInputMatlab()

```
void parseInputMatlab (
    parameters * param,
    const char * paramString,
    bool trainingPhase,
    const char * inputFileName,
    const char * modelName )
```

Parses the user input and modifies parameter settings as necessary.

Parameters

out	<i>param</i>	Parameter object modified by user input.
in	<i>paramString</i>	User-provided parameter string, can be NULL in which case default parameters are used..
in	<i>trainingPhase</i>	Indicator if training or testing phase.
in	<i>inputFileName</i>	User-provided filename with input data (if NULL no check of filename validity).
in	<i>modelName</i>	User-provided filename with learned model (if NULL no check of filename validity).

Definition at line 1249 of file budgetedSVM_matlab.cpp.

```
1250 {
1251     int pos = 0, tempPos = 0, len;
1252     char str[256];
```

```

1253     vector <char> option;
1254     vector <float> value;
1255     FILE *pFile = NULL;
1256
1257     if (paramString == NULL)
1258         len = 0;
1259     else
1260         len = (int) strlen(paramString);
1261
1262     // check if the input data file exists only if input data filename is provided
1263     if (inputFileName)
1264     {
1265         if (!readableFileExists(inputFileName))
1266         {
1267             sprintf(str, "Can't open input file %s!\n", inputFileName);
1268             mexErrMsgTxt(str);
1269         }
1270     }
1271
1272     while (pos < len)
1273     {
1274         if (paramString[pos++] == '-')
1275         {
1276             option.push_back(paramString[pos]);
1277             pos += 2;
1278
1279             tempPos = 0;
1280             while ((paramString[pos] != ' ') && (paramString[pos] != '\0'))
1281             {
1282                 str[tempPos++] = paramString[pos++];
1283             }
1284             str[tempPos++] = '\0';
1285             value.push_back((float) atof(str));
1286         }
1287     }
1288
1289     if (trainingPhase)
1290     {
1291         // check if the model file exists only if model filename is provided
1292         if (modelFileName)
1293         {
1294             pFile = fopen(modelFileName, "w");
1295             if (pFile == NULL)
1296             {
1297                 sprintf(str, "Can't create model file %s!\n", modelFileName);
1298                 mexErrMsgTxt(str);
1299             }
1300             else
1301             {
1302                 fclose(pFile);
1303                 pFile = NULL;
1304             }
1305         }
1306
1307         // modify parameters
1308         for (unsigned int i = 0; i < option.size(); i++)
1309         {
1310             switch (option[i])
1311             {
1312                 case 'A':
1313                     (*param).ALGORITHM = (unsigned int) value[i];
1314                     if ((*param).ALGORITHM > 4)
1315                     {
1316                         sprintf(str, "Input parameter '-A %d' out of bounds!\nRun 'budgetedsvm_train()'
for help.", (*param).ALGORITHM);
1317                         mexErrMsgTxt(str);
1318                     }
1319                     break;
1320                 case 'e':
1321                     (*param).NUM_EPOCHS = (unsigned int) value[i];
1322                     break;
1323                 case 's':
1324                     (*param).NUM_SUBEPOCHS = (unsigned int) value[i];
1325                     break;
1326                 case 'k':
1327                     (*param).K_PARAM = (unsigned int) value[i];
1328                     break;
1329                 case 'c':
1330                     (*param).C_PARAM = value[i];
1331                     if ((*param).C_PARAM <= 0.0)
1332                     {
1333                         sprintf(str, "Input parameter '-c' should be a positive real number!\nRun
'budgetedsvm_train()' for help.");
1334                         mexErrMsgTxt(str);
1335                     }
1336                     break;
1337                 case 'L':

```

```

1338         (*param).LAMBDA_PARAM = (double) value[i];
1339         if ((*param).LAMBDA_PARAM <= 0.0)
1340         {
1341             sprintf(str, "Input parameter '-L' should be a positive real number!\nRun
'budgetedsvm_train()' for help.");
1342             mexErrMsgTxt(str);
1343         }
1344         break;
1345
1346         case 'K':
1347             (*param).KERNEL = (unsigned int) value[i];
1348             break;
1349
1350         case 'g':
1351             (*param).KERNEL_GAMMA_PARAM = (long double) value[i];
1352             if ((*param).KERNEL_GAMMA_PARAM <= 0.0)
1353             {
1354                 sprintf(str, "Input parameter '-g' should be a positive real number!\nRun
'budgetedsvm_train()' for help.");
1355                 mexErrMsgTxt(str);
1356             }
1357             break;
1358
1359         case 'd':
1360             (*param).KERNEL_DEGREE_PARAM = (double) value[i];
1361             if ((*param).KERNEL_DEGREE_PARAM <= 0.0)
1362             {
1363                 sprintf(str, "Input parameter '-d' should be a positive real number!\nRun
'budgetedsvm_train()' for help.\n");
1364                 mexErrMsgTxt(str);
1365             }
1366             break;
1367
1368         case 'i':
1369             (*param).KERNEL_COEF_PARAM = (double) value[i];
1370             break;
1371
1372         case 'm':
1373             (*param).MAINTENANCE_SAMPLING_STRATEGY = (unsigned int) value[i];
1374             break;
1375
1376         case 'b':
1377             (*param).BIAS_TERM = (double) value[i];
1378             break;
1379         case 'v':
1380             (*param).VERBOSE = (value[i] != 0);
1381             break;
1382         case 'r':
1383             (*param).RANDOMIZE = (value[i] != 0);
1384             break;
1385         case 'B':
1386             (*param).BUDGET_SIZE = (unsigned int) value[i];
1387             if ((*param).BUDGET_SIZE < 1)
1388             {
1389                 sprintf(str, "Input parameter '-B' should be a positive integer!\nRun
'budgetedsvm_train()' for help.");
1390                 mexErrMsgTxt(str);
1391             }
1392             break;
1393         case 'D':
1394             // a user explicitly assigns dimensionality only if data set is given in .txt file,
1395             // otherwise dimensionality is found directly from Matlab, no need for a user to specify it
1396             if (inputFileName)
1397             {
1398                 (*param).DIMENSION = (unsigned int) value[i];
1399             }
1400             else
1401             {
1402                 //sprintf(str, "Warning, if data loaded to Matlab no need to set '-D %d'
1403                 //option.\nRun 'budgetedsvm_train()' for help.\n", (int) value[i]);
1404                 //mexPrintf(str);
1405             }
1406             break;
1407
1408         case 'z':
1409             (*param).CHUNK_SIZE = (unsigned int) value[i];
1410             if ((*param).CHUNK_SIZE < 1)
1411             {
1412                 sprintf(str, "Input parameter '-z' should be an integer larger than 0!\nRun
'budgetedsvm_train()' for help.");
1413                 mexErrMsgTxt(str);
1414             }
1415             break;
1416         case 'w':
1417             (*param).CHUNK_WEIGHT = (unsigned int) value[i];
1418             if ((*param).CHUNK_WEIGHT < 1)
1419             {

```

```

1418         sprintf(str, "Input parameter '-w' should be an integer larger than 0!\nRun
'budgetedsvm_train()' for help.");
1419         mexErrMsgTxt(str);
1420     }
1421     break;
1422     case 'S':
1423         (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
1424         break;
1425     case 'C':
1426         (*param).CLONE_PROBABILITY = value[i];
1427         if ((*param).CLONE_PROBABILITY < 0)
1428             (*param).CLONE_PROBABILITY = 0;
1429         else if ((*param).CLONE_PROBABILITY > 1)
1430             (*param).CLONE_PROBABILITY = 1;
1431         break;
1432     case 'Y':
1433         (*param).CLONE_PROBABILITY_DECAY = value[i];
1434         if ((*param).CLONE_PROBABILITY_DECAY < 0)
1435             (*param).CLONE_PROBABILITY_DECAY = 0;
1436         else if ((*param).CLONE_PROBABILITY_DECAY > 1)
1437             (*param).CLONE_PROBABILITY_DECAY = 1;
1438         break;
1439     default:
1440         sprintf(str, "Error, unknown input parameter '%c'!\nRun 'budgetedsvm_train()' for
help.", option[i]);
1441         mexErrMsgTxt(str);
1442         break;
1443     }
1444 }
1445 }
1446
1447 // for BSGD, when we use merging budget maintenance strategy then only Gaussian kernel can be
used,
1448 // due to the nature of merging; here check if user specified some other kernel while merging
1449 if (((*param).ALGORITHM == BSGD) && ((*param).KERNEL != KERNEL_FUNC_GAUSSIAN) &&
(((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE))
1450 {
1451     mexPrintf("Warning, BSGD with merging strategy can only use Gaussian kernel!\nKernel
function switched to Gaussian.\n");
1452     (*param).KERNEL = KERNEL_FUNC_GAUSSIAN;
1453 }
1454
1455 // check the MAINTENANCE_SAMPLING_STRATEGY validity
1456 if ((*param).ALGORITHM == LLSVM)
1457 {
1458     if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 2)
1459     {
1460         // 0 - random removal, 1 - k-means, 2 - k-medoids
1461         sprintf(str, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm_train()' for
help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
1462         mexErrMsgTxt(str);
1463     }
1464 }
1465 else if ((*param).ALGORITHM == BSGD)
1466 {
1467     if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 1)
1468     {
1469         // 0 - smallest removal, 1 - merging
1470         sprintf(str, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm_train()' for
help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
1471         mexErrMsgTxt(str);
1472     }
1473 }
1474
1475 // no bias term for LLSVM and BSGD functions
1476 if (((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD))
1477     (*param).BIAS_TERM = 0.0;
1478
1479 if ((*param).VERBOSE)
1480 {
1481     mexPrintf("*** Training started with the following parameters:\n");
1482     switch ((*param).ALGORITHM)
1483     {
1484         case PEGASOS:
1485             mexPrintf("Algorithm \t\t\t\t: Pegasos\n");
1486             break;
1487         case AMM_ONLINE:
1488             mexPrintf("Algorithm \t\t\t\t: AMM online\n");
1489             break;
1490         case AMM_BATCH:
1491             mexPrintf("Algorithm \t\t\t\t: AMM batch\n");
1492             break;
1493         case BSGD:
1494             mexPrintf("Algorithm \t\t\t\t: BSGD\n");
1495             break;
1496         case LLSVM:
1497             mexPrintf("Algorithm \t\t\t\t: LLSVM\n");

```

```

1498         break;
1499     }
1500
1501     if ((*param).ALGORITHM == PEGASOS) || ((*param).ALGORITHM == AMM_BATCH) ||
1502     ((*param).ALGORITHM == AMM_ONLINE))
1503     {
1504         mexPrintf("Lambda parameter \t\t: %f\n", (*param).LAMBDA_PARAM);
1505         mexPrintf("Bias term \t\t\t: %f\n", (*param).BIAS_TERM);
1506         if ((*param).ALGORITHM != PEGASOS)
1507         {
1508             mexPrintf("Pruning frequency k \t: %d\n", (*param).K_PARAM);
1509             mexPrintf("Pruning threshold c \t: %f\n", (*param).C_PARAM);
1510             mexPrintf("Num. weights per class\t: %d\n", (*param).BUDGET_SIZE);
1511             mexPrintf("Number of epochs \t\t: %d\n\n", (*param).NUM_EPOCHS);
1512         }
1513         else
1514             mexPrintf("\n");
1515     }
1516     else if ((*param).ALGORITHM == BSGD) || ((*param).ALGORITHM == LLSVM)
1517     {
1518         if ((*param).ALGORITHM == BSGD)
1519         {
1520             mexPrintf("Number of epochs \t\t\t: %d\n", (*param).NUM_EPOCHS);
1521             mexPrintf("Size of the budget \t\t\t: %d\n", (*param).BUDGET_SIZE);
1522             if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_REMOVE)
1523                 mexPrintf("Maintenance strategy \t\t: smallest removal\n");
1524             else if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE)
1525                 mexPrintf("Maintenance strategy \t\t: merging\n");
1526             else
1527                 mexErrMsgTxt("Error, unknown budget maintenance set. Run 'budgetedsvm_train()'
1528                 for help.\n");
1529
1530             mexPrintf("Lambda regularization param.: %f\n", (*param).LAMBDA_PARAM);
1531         }
1532         else if ((*param).ALGORITHM == LLSVM)
1533         {
1534             switch ((*param).MAINTENANCE_SAMPLING_STRATEGY)
1535             {
1536                 case LANDMARK_SAMPLE_RANDOM:
1537                     mexPrintf("Landmark sampling \t\t\t: random sampling\n");
1538                     break;
1539
1540                 case LANDMARK_SAMPLE_KMEANS:
1541                     mexPrintf("Landmark sampling \t\t\t: k-means initialization\n");
1542                     break;
1543
1544                 case LANDMARK_SAMPLE_KMEDOIDS:
1545                     mexPrintf("Landmark sampling \t\t\t: k-medoids initialization\n");
1546                     break;
1547
1548                 default:
1549                     mexErrMsgTxt("Error, unknown landmark sampling set. Run
1550                     'budgetedsvm_train()' for help.\n");
1551                     break;
1552             }
1553             mexPrintf("Number of landmark points \t: %d\n", (*param).BUDGET_SIZE);
1554             mexPrintf("Lambda regularization param.: %f\n", (*param).LAMBDA_PARAM);
1555         }
1556     }
1557
1558     // print common parameters
1559     switch ((*param).KERNEL)
1560     {
1561         case KERNEL_FUNC_GAUSSIAN:
1562             mexPrintf("Gaussian kernel used \t\t:  $K(x, y) = \exp(-0.5 * \gamma * ||x -$ 
1563              $y||^2)\n"$ );
1564             if ((*param).KERNEL_GAMMA_PARAM != 0.0)
1565             {
1566                 sprintf(str, "Gaussian kernel width \t\t: %f\n\n",
1567                 (*param).KERNEL_GAMMA_PARAM);
1568                 mexPrintf(str);
1569             }
1570             else
1571                 mexPrintf("Gaussian kernel width \t\t: 1 / DIMENSIONALITY\n\n");
1572             break;
1573
1574         case KERNEL_FUNC_EXPONENTIAL:
1575             mexPrintf("Exponential kernel used \t:  $K(x, y) = \exp(-0.5 * \gamma * ||x -$ 
1576              $y||)\n"$ );
1577             if ((*param).KERNEL_GAMMA_PARAM != 0.0)
1578             {
1579                 sprintf(str, "Exponential kernel width \t: %f\n\n",
1580                 (*param).KERNEL_GAMMA_PARAM);
1581                 mexPrintf(str);
1582             }
1583             else
1584                 mexPrintf("Exponential kernel width \t: 1 / DIMENSIONALITY\n\n");
1585             break;
1586     }

```



```

1578
1579         case KERNEL_FUNC_POLYNOMIAL:
1580             sprintf(str, "Polynomial kernel used \t\t: K(x, y) = (x^T * y +
1581             %.2f)^%.2f\n", (*param).KERNEL_COEF_PARAM, (*param).KERNEL_DEGREE_PARAM);
1582             mexPrintf(str);
1583             break;
1584
1585         case KERNEL_FUNC_SIGMOID:
1586             sprintf(str, "Sigmoid kernel used \t\t: K(x, y) = tanh(%.2f * x^T * y +
1587             %.2f)\n", (*param).KERNEL_DEGREE_PARAM, (*param).KERNEL_COEF_PARAM);
1588             mexPrintf(str);
1589             break;
1590
1591         case KERNEL_FUNC_LINEAR:
1592             mexPrintf("Linear kernel used \t\t\t: K(x, y) = (x^T * y)\n");
1593             break;
1594
1595         case KERNEL_FUNC_USER_DEFINED:
1596             mexPrintf("User-defined kernel function used.\n");
1597             break;
1598
1599         default:
1600             sprintf(str, "Input parameter '-K %d' out of bounds!\nRun 'budgetedsvm_train()'
1601             for help.\n", (*param).KERNEL);
1602             mexErrMsgTxt(str);
1603             break;
1604     }
1605     mexEvalString("drawnow;");
1606
1607     // if inputs to training phase are .txt files, then also increase dimensionality due to added
1608     // NOTE that we do not execute this part if inputs are Matlab variables, as we still do not
1609     // know the dimensionality, therefore BIAS_TERM and
1610     // KERNEL_GAMMA_PARAM are adjusted in budgetedDataMatlab::readDataFromMatlab() function, after
1611     // we find out the dimensionality of the considered data set
1612     if (inputFileName)
1613     {
1614         // signal error if a user wants to use an RBF kernel, but didn't specify either data
1615         // dimension or kernel width
1616         if (((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD)) && ((*param).KERNEL ==
1617         KERNEL_FUNC_GAUSSIAN) || ((*param).KERNEL == KERNEL_FUNC_EXPONENTIAL))
1618         {
1619             if ((*param).KERNEL_GAMMA_PARAM == 0.0) && ((*param).DIMENSION == 0)
1620             {
1621                 // this means that both RBF kernel width and dimension were not set by the user in
1622                 // the input string to the toolbox
1623                 // since in this case the default value of RBF kernel is 1/dimensionality, report
1624                 // error to the user
1625                 mexErrMsgTxt("Error, RBF kernel in use, please set either kernel width or
1626                 dimensionality!\nRun 'budgetedsvm_train()' for help.\n");
1627             }
1628         }
1629
1630         // increase dimensionality if bias term included
1631         if ((*param).BIAS_TERM != 0.0)
1632         {
1633             (*param).DIMENSION++;
1634         }
1635
1636         // set gamma to default value of dimensionality
1637         if ((*param).KERNEL_GAMMA_PARAM == 0.0)
1638             (*param).KERNEL_GAMMA_PARAM = 1.0 / (double) (*param).DIMENSION;
1639     }
1640
1641     else
1642     {
1643         // check if the model file exists only if model filename is provided
1644         if (modelFileName)
1645         {
1646             if (!readableFileExists(modelFileName))
1647             {
1648                 sprintf(str, "Can't open model file %s!\n", modelFileName);
1649                 mexErrMsgTxt(str);
1650             }
1651         }
1652
1653         // modify parameters
1654         for (unsigned int i = 0; i < option.size(); i++)
1655         {
1656             switch (option[i])
1657             {
1658                 /*case 'p':
1659                     (*param).SAVE_PRED = (value[i] != 0);
1660                     break;*/
1661                 case 'v':

```

```

1654         (*param).VERBOSE = (value[i] != 0);
1655         break;
1656
1657         case 'z':
1658             (*param).CHUNK_SIZE = (unsigned int) value[i];
1659             if ((*param).CHUNK_SIZE < 1)
1660             {
1661                 sprintf(str, "Input parameter '-z' should be an integer larger than 0!\nRun
'budgetedsvm_train()' for help.");
1662                 mexErrMsgTxt(str);
1663             }
1664             break;
1665         case 'w':
1666             (*param).CHUNK_WEIGHT = (unsigned int) value[i];
1667             if ((*param).CHUNK_WEIGHT < 1)
1668             {
1669                 sprintf(str, "Input parameter '-w' should be an integer larger than 0!\nRun
'budgetedsvm_train()' for help.");
1670                 mexErrMsgTxt(str);
1671             }
1672             break;
1673         case 's':
1674             (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
1675             break;
1676
1677         default:
1678             sprintf(str, "Error, unknown input parameter '%c'!\nRun 'budgetedsvm_predict()'
for help.", option[i]);
1679             mexErrMsgTxt(str);
1680             break;
1681     }
1682 }
1683
1684 /*if ((*param).VERBOSE)
1685 {
1686     mexPrintf("\n*** Testing with the following parameters:\n");
1687     switch ((*param).ALGORITHM)
1688     {
1689         case PEGASOS:
1690             mexPrintf("Algorithm: \t\t\t\tPEGASOS\n");
1691             break;
1692         case AMM_ONLINE:
1693             mexPrintf("Algorithm: \t\t\t\tAMM online\n");
1694             break;
1695         case AMM_BATCH:
1696             mexPrintf("Algorithm: \t\t\t\tAMM batch\n");
1697             break;
1698         case BSGD:
1699             mexPrintf("Algorithm: \t\t\t\tBSGD\n");
1700             break;
1701     }
1702
1703     if ((*param).ALGORITHM == PEGASOS || ((*param).ALGORITHM == AMM_BATCH) ||
(*param).ALGORITHM == AMM_ONLINE)
1704     {
1705         mexPrintf("Bias term: \t\t\t\t%f\n", (*param).BIAS_TERM);
1706     }
1707     else if ((*param).ALGORITHM == BSGD)
1708     {
1709         mexPrintf("Gaussian kernel width: \t\t\t\t%f\n", (*param).GAMMA_PARAM);
1710     }
1711     mexEvalString("drawnow;");
1712 }*/
1713 }
1714
1715 setPrintErrorStringFunction(&printErrorStringMatlab);
1716 if ((*param).VERBOSE)
1717     setPrintStringFunction(&printStringMatlab);
1718 else
1719     setPrintStringFunction(NULL);
1720 }

```

6.1.2.3 printErrorStringMatlab()

```

void printErrorStringMatlab (
    const char * s )

```

Prints error string to Matlab, used to modify callback found in [budgetedSVM.cpp](#).

in	s	Text to be printed.
----	---	---------------------

```

1110 {
1111     mexErrMsgTxt(s);
1112 }

```

in	s	Text to be printed.
----	---	---------------------

```
1100 {
1101     mexPrintf(s);
1102     mexEvalString("drawnow;");
1103 }
```

in	<i>trainingPhase</i>	Indicator if training or testing phase.
in	<i>param</i>	Parameter object modified by user input.

[illegible]

[illegible]

6.2 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSV↵ M/src/bsgd.h File Reference

Defines classes and functions used for training and testing of BSGD (Budgeted Stochastic Gradient Descent) algorithm.

Classes

- class `budgetedVectorBSGD`

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for BSGD algorithm.

- class `budgetedModelBSGD`

Class which holds the BSGD model (comprising the support vectors stored as `budgetedVectorBSGD`), and implements methods to load BSGD model from and save BSGD model to text file.

Functions

- void `trainBSGD` (`budgetedData` *trainData, `parameters` *param, `budgetedModelBSGD` *model)
Train BSGD.
- float `predictBSGD` (`budgetedData` *testData, `parameters` *param, `budgetedModelBSGD` *model, vector< int > *labels=NULL, vector< float > *scores=NULL)
Given a BSGD model, predict the labels of testing data.

6.2.1 Detailed Description

Defines classes and functions used for training and testing of BSGD (Budgeted Stochastic Gradient Descent) algorithm.

6.2.2 Function Documentation

6.2.2.1 predictBSGD()

```
float predictBSGD (
    budgetedData * testData,
    parameters * param,
    budgetedModelBSGD * model,
    vector< int > * labels,
    vector< float > * scores )
```

Given a BSGD model, predict the labels of testing data.

Parameters

in	<i>testData</i>	Input test data.
in	<i>param</i>	The parameters of the algorithm.
in	<i>model</i>	Trained BSGD model.
out	<i>labels</i>	Vector of predicted labels.
out	<i>scores</i>	Vector of scores of the winning labels.

Returns

Testing set error rate.

Given the learned BSGD model, the function computes the predictions on the testing data, outputting the predicted labels and the error rate.

Definition at line 513 of file `bsgd.cpp`.

```
514 {
515     unsigned long timeCalc = 0, start;
516     unsigned int i, N, err = 0, total = 0;
517     long double fx, maxFx, tempSqrNorm = 0.0, *tempArray;
518     bool stillChunksLeft = true;
519     char text[1024];
520     unsigned int y;
```

```

521     budgetedVectorBSGD *currentDataPoint = NULL;
522     long double *classMaxScores = new long double[(testData->yLabels).size()];
523
524     // this tempArray is used when calculating all class scores, to avoid repeated computations of the
    same kernel
525     tempArray = new long double[(*(model->modelBSGD)).size()];
526     for (i = 0; i < (*(model->modelBSGD)).size(); i++)
527         tempArray[i] = 0.0;
528
529     while (stillChunksLeft)
530     {
531         stillChunksLeft = testData->readChunk((*param).CHUNK_SIZE);
532         (*param).updateVerySparseDataParameter(testData->getSparsity());
533
534         N = testData->N;
535         total += N;
536         start = clock();
537
538         for (unsigned int r = 0; r < N; r++)
539         {
540             if ((*param).VERY_SPARSE_DATA)
541             {
542                 // since we are computing kernels using vectors directly from the budgetedData, we need
    square norm of the vector to speed-up
543                 // computations, here we compute it just once; no need to do it in non-sparse case,
    since this norm can be retrieved directly
544                 // from budgetedVector
545                 tempSqrNorm = testData->getVectorSqrL2Norm(r, param);
546             }
547             else
548             {
549                 // create the budgetedVector using the vector from budgetedData, to be used in kernel
    computations below
550                 currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT,
    (unsigned int) (testData->yLabels).size());
551                 currentDataPoint->createVectorUsingDataPoint(testData, r, param);
552             }
553
554             y = 0;
555             maxFx = -INF;
556             for (i = 0; i < (*(model->modelBSGD)).size(); i++)
557                 tempArray[i] = 0.0;
558
559             for (unsigned int k = 0; k < (testData->yLabels).size(); k++)
560             {
561                 classMaxScores[k] = -INF;
562                 fx = 0;
563                 for (unsigned int i = 0; i < (*(model->modelBSGD)).size(); i++)
564                 {
565                     if ((*model->modelBSGD)[i]->alphas[k] != 0)
566                     {
567                         if (tempArray[i] == 0.0)
568                         {
569                             if ((*param).VERY_SPARSE_DATA)
570                             {
571                                 // directly compute kernel from the trainData
572                                 tempArray[i] = ((*model).modelBSGD)[i]->computeKernel(r, testData,
    param, tempSqrNorm);
573                             }
574                             else
575                             {
576                                 // compute kernel from currentDataPoint object
577                                 tempArray[i] = (*model->modelBSGD)[i]->computeKernel(currentDataPoint,
    param);
578                             }
579                         }
580                         fx += ((*model->modelBSGD)[i]->alphas[k] * tempArray[i]);
581                     }
582                 }
583
584                 if (fx > maxFx)
585                 {
586                     maxFx = fx;
587                     y = k;
588                 }
589
590                 if (fx > classMaxScores[k])
591                     classMaxScores[k] = fx;
592             }
593
594             if (!(*param).VERY_SPARSE_DATA)
595             {
596                 // if sparse then no need for this, since we didn't even create currentDataPoint
597                 delete currentDataPoint;
598                 currentDataPoint = NULL;
599             }
600

```

```

601         if (y != testData->al[r])
602             err++;
603
604         // save predicted label, will be sent to output
605         if (labels)
606             (*labels).push_back((int) (testData->yLabels)[y]);
607         // ... and the scores
608         if (scores)
609         {
610             // for BSGD the output score is the difference between winning and the second best score
611             long double secondBestScore = -INF;
612             for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
613             {
614                 if (i == y)
615                     continue;
616
617                 if (secondBestScore < classMaxScores[i])
618                     secondBestScore = classMaxScores[i];
619             }
620             (*scores).push_back((float) (maxFx - secondBestScore));
621         }
622     }
623
624     timeCalc += clock() - start;
625
626     if ((*param).VERBOSE) && (N > 0)
627     {
628         sprintf(text, "Number of examples processed: %d\n", total);
629         svmPrintString(text);
630     }
631 }
632 testData->flushData();
633 delete [] tempArray;
634 delete[] classMaxScores;
635
636 if ((*param).VERBOSE)
637 {
638     sprintf(text, "*** Testing completed in %5.3f seconds\n*** Testing error rate: %3.2f\n",
639             (double)timeCalc / (double)CLOCKS_PER_SEC, 100.0 * (float)err / (float)total);
640     svmPrintString(text);
641 }
642 return (float) (100.0 * (float)err / (float)total);
643 }

```

6.2.2.2 trainBSGD()

```

void trainBSGD (
    budgetedData * trainData,
    parameters * param,
    budgetedModelBSGD * model )

```

Train BSGD.

Parameters

in	<i>trainData</i>	Input training data.
in	<i>param</i>	The parameters of the algorithm.
in, out	<i>model</i>	Initial BSGD model.

The function trains BSGD model, given input data, the initial model (most often zero-weight model), and the parameters of the model.

Definition at line 653 of file bsgd.cpp.

```

654 {
655     unsigned long timeCalc = 0, start;
656     long double fxValue, fxValue1, fxValue2, maxFx, *tempArray, alphaSmallest = 0.0, tempLongDouble = 0.0;

```



```

657     unsigned int i1, i2 = 0, t, temp, countDel = 0, numClasses = 0, numSVs = 0, numIter = 0, N,
deleteWeight = 0;
658     bool stillChunksLeft = true;
659     char text[1024];
660     unsigned int i, k, ot; //iterators
661     budgetedVectorBSGD *currentDataPoint = NULL;
662     int indexOfSameVector = -1; // this variable keeps the index of the *exact same* vector in the SV
set, when compared to input point.
663     // so when we observe budget overflow we merge these two if merging
strategy is set
664
665     // this tempArray is used when calculating all class scores and runner-up, to avoid repeated
computations of the same kernel
666     tempArray = new long double[(*param).BUDGET_SIZE];
667     for (i = 0; i < (*param).BUDGET_SIZE; i++)
668         tempArray[i] = 0.0;
669
670     for (unsigned int epoch = 0; epoch < (*param).NUM_EPOCHS; epoch++)
671     {
672         //Calculate
673         stillChunksLeft = true;
674         while (stillChunksLeft)
675         {
676             stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);
677
678             // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
679             // (of course, in the case of AMM, speeds up linear kernel computation)
680             (*param).updateVerySparseDataParameter(trainData->getSparsity());
681
682             // compute observed data dimensionality, where we also account for possible bias term, and
check if
683             // we need to expand the current model weights if some new data dimensions were found
during loading
684             temp = trainData->getDataDimensionality() + (int) (param->BIAS_TERM != 0.0);
685             if ((*param).DIMENSION < temp)
686             {
687                 /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION,
temp);
688                 svmPrintString(text);*/
689                 (*model).extendDimensionalityOfModel(temp, param);
690
691                 // update the dimensionality
692                 (*param).DIMENSION = temp;
693             }
694
695             N = trainData->N;
696             if (numIter == 0)
697                 numClasses = (unsigned int) trainData->yLabels.size();
698             else if (numClasses != (unsigned int) trainData->yLabels.size())
699             {
700                 // if in the earlier chunks some class wasn't observed, it could happen with small
chunks or unbalanced classes;
701                 // just add new zero alphas for the new classes to each support vector
702                 for (unsigned int i = 0; i < numSVs; i++)
703                     for (unsigned int k = 0; k < (trainData->yLabels.size() - numClasses); k++)
704                         ((*model).modelBSGD)[i]->alphas.push_back(0.0);
705                 numClasses = (unsigned int) trainData->yLabels.size();
706             }
707
708             // randomize
709             vector<unsigned int> tv(N, 0);
710             for (unsigned int ti = 0; ti < N; ti++)
711             {
712                 tv[ti] = ti;
713             }
714             if ((*param).RANDOMIZE)
715                 random_shuffle(tv.begin(), tv.end());
716
717             start = clock();
718             for (ot = 0; ot < N; ot++)
719             {
720                 t = tv[ot];
721                 numIter++;
722
723                 // initialize the first weight
724                 if (numIter == 1)
725                 {
726                     currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT,
numClasses);
727                     currentDataPoint->createVectorUsingDataPoint(trainData, t, param);
728
729                     i1 = trainData->al[t];
730                     i2 = (i1 + 1) % numClasses;
731                     currentDataPoint->alphas[i1] = 1.0;
732                     currentDataPoint->alphas[i2] = -1.0;
733
734                     ((*model).modelBSGD).push_back(currentDataPoint);

```

```

735         currentDataPoint = NULL;
736         numSVs++;
737         continue;
738     }
739
740     // calculate all class scores and runner-up
741     i1 = trainData->al[t];
742     fxValue1 = 0.0;
743     fxValue2 = 0.0;
744     maxFx = -INF;
745     for (i = 0; i < numSVs; i++)
746         tempArray[i] = 0.0;
747
748     if ((*param).VERY_SPARSE_DATA)
749     {
750         // since we are computing kernels using vectors directly from the budgetedData, we
751         // need square norm of the vector to speed-up
752         // computations, here we compute it just once; no need to do it in non-sparse case,
753         // since this norm can be retrieved directly
754         // from budgetedVector
755         tempLongDouble = trainData->getVectorSqrL2Norm(t, param);
756     }
757     else
758     {
759         // create the budgetedVector using the vector from budgetedData, to be used in
760         // gaussianKernel() method below
761         currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT,
762         numClasses);
763         currentDataPoint->createVectorUsingDataPoint(trainData, t, param);
764     }
765
766     indexOfSameVector = -1;
767     for (k = 0; k < numClasses; k++)
768     {
769         fxValue = 0.0;
770         for (i = 0; i < numSVs; i++)
771         {
772             if ((*((*model).modelBSGD))[i]->alphas[k] != 0)
773             {
774                 // calculate the kernel only if not computed earlier
775                 if (tempArray[i] == 0.0)
776                 {
777                     if ((*param).VERY_SPARSE_DATA)
778                     {
779                         // directly compute kernel from the trainData
780                         tempArray[i] = ((*((*model).modelBSGD))[i]->computeKernel(t,
781                         trainData, param, tempLongDouble));
782                     }
783                     else
784                     {
785                         // check if the two vectors are identical, if they are then we
786                         // consider these two vectors when budget overflow
787                         // happens (we round to 8th digit due to round-off errors in
788                         // floating-point representation, observed in practice)
789                         if ((int) ((*((*model).modelBSGD))[i]->gaussianKernel(t, trainData,
790                         param, tempLongDouble) * 100000000.0 + 0.5) == 100000000)
791                         {
792                             indexOfSameVector = i;
793                         }
794                     }
795                 }
796                 else
797                 {
798                     // compute kernel from currentDataPoint object
799                     tempArray[i] =
800                     ((*((*model).modelBSGD))[i]->computeKernel(currentDataPoint, param);
801
802                     // check if the two vectors are identical, if they are then we
803                     // consider these two vectors when budget overflow
804                     // happens (we round to 8th digit due to round-off errors in
805                     // floating-point representation, observed in practice)
806                     if ((int) ((*((*model).modelBSGD))[i]->gaussianKernel(currentDataPoint, param) * 100000000.0 + 0.5) ==
807                     100000000)
808                     {
809                         indexOfSameVector = i;
810                     }
811                 }
812             }
813             fxValue += ((*((*model).modelBSGD))[i]->alphas[k] * tempArray[i]);
814         }
815     }
816
817     if (k == i1)
818         fxValue1 = fxValue;
819     else if (fxValue > maxFx)
820     {
821         maxFx = fxValue;
822         i2 = k;
823     }

```

```

809         }
810         fxValue2 = maxFx;
811
812         // downweight all the weights
813         for (i = 0; i < numSVs; i++)
814             ((*model).modelBSGD)[i]->downgrade(numIter);
815
816         if (1.0 + fxValue2 - fxValue1 > 0.0)
817         {
818             if ((*param).VERY_SPARSE_DATA)
819             {
820                 // only do this if data is sparse, since if non-sparse than we already have
821                 currentDataPoint initialized
822                 // from the code before the loop in which we computed kernels
823                 currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION,
824                     (*param).CHUNK_WEIGHT, numClasses);
825                 currentDataPoint->createVectorUsingDataPoint(trainData, t, param);
826
827                 // add an SV
828                 currentDataPoint->alphas[i1] = 1.0 / ((long double)numIter *
829                     (*param).LAMBDA_PARAM);
830                 currentDataPoint->alphas[i2] = -1.0 / ((long double)numIter *
831                     (*param).LAMBDA_PARAM);
832                 ((*model).modelBSGD).push_back(currentDataPoint);
833                 currentDataPoint = NULL;
834                 numSVs++;
835
836                 // if over the budget, maintain the budget
837                 if (numSVs > (*param).BUDGET_SIZE)
838                 {
839                     switch ((*param).MAINTENANCE_SAMPLING_STRATEGY)
840                     {
841                         case BUDGET_MAINTAIN_REMOVE:
842                             // removal of random support vector
843                             if (indexOfSameVector == -1)
844                             {
845                                 // so there are no two identical vectors, remove the smallest one
846                                 alphaSmallest = INF;
847                                 for (i = 0; i < numSVs; i++)
848                                 {
849                                     // compute product between norm of alpha vector and a
850                                     self-kernel
851                                     tempLongDouble = ((*model).modelBSGD)[i]->alphaNorm() *
852                                     ((*model).modelBSGD)[i]->computeKernel((*model).modelBSGD)[i], param);
853                                     if (alphaSmallest > tempLongDouble)
854                                     {
855                                         alphaSmallest = tempLongDouble;
856                                         deleteWeight = i;
857                                     }
858                                 }
859                             }
860                             else
861                             {
862                                 // since there is already an identical vector in the SV set, remove
863                                 the newly added vector
864                                 deleteWeight = (*param).BUDGET_SIZE;
865                             }
866                             delete ((*model).modelBSGD)[deleteWeight];
867                             ((*model).modelBSGD).erase((*model).modelBSGD).begin() +
868                             deleteWeight);
869                             break;
870                         case BUDGET_MAINTAIN_MERGE:
871                             // merging of two SVs
872                             long double kMax, kZ1, kZ2;
873                             unsigned int merge1, merge2;
874
875                             if (indexOfSameVector == -1)
876                             {
877                                 // so there are no two identical vectors. find the one with smallest
878                                 alpha
879                                 // here we look for who to merge
880                                 merge1 = 0;
881                                 for (i = 0; i < numSVs; i++)
882                                 {
883                                     tempLongDouble = ((*model).modelBSGD)[i]->alphaNorm();
884                                     if ((alphaSmallest > tempLongDouble) || (i == 0))
885                                     {
886                                         alphaSmallest = tempLongDouble;
887                                         merge1 = i;
888                                     }
889                                 }
890
891                                 // find with who to merge, as well as other useful information
892                                 detailed in the definition of computeKmax() found in this file

```

```

886         long double* returnValues = computeKmax((*model).modelBSGD, merge1,
887 param);
888         kMax = (*returnValues);
889         kZ1 = (*returnValues + 1);
890         kZ2 = (*returnValues + 2);
891         merge2 = (unsigned int) (*returnValues + 3);
892         delete [] returnValues;
893
894         // find z, the new support vector
895         ((*model).modelBSGD)[merge1]->updateSV ((*model).modelBSGD)[merge2], kMax);
896         for (unsigned int k = 0; k < numClasses; k++)
897             ((*model).modelBSGD)[merge1]->alphas[k] =
898             ((*model).modelBSGD)[merge2]->alphas[k] * kZ1 + ((*model).modelBSGD)[merge2]->alphas[k] * kZ2;
899         }
900         else
901         {
902             // in this case the two merging vectors are identical, so we simply
903             add up their alphas and there is no need for moving the vector
904             merge1 = indexOfSameVector;
905             merge2 = (*param).BUDGET_SIZE;
906             for (unsigned int k = 0; k < numClasses; k++)
907                 ((*model).modelBSGD)[merge1]->alphas[k] =
908                 ((*model).modelBSGD)[merge1]->alphas[k] + ((*model).modelBSGD)[merge2]->alphas[k];
909             }
910             // delete 'merge2', not needed anymore
911             delete ((*model).modelBSGD)[merge2];
912             ((*model).modelBSGD).erase ((*model).modelBSGD).begin() + merge2);
913             break;
914         }
915         numSVs--;
916         countDel++;
917     }
918     }
919     else
920     {
921         if ((*param).VERY_SPARSE_DATA)
922         {
923             // if sparse data then no need for this part, since we didn't even create
924             currentDataPoint
925             delete currentDataPoint;
926             currentDataPoint = NULL;
927         }
928     }
929     }
930     timeCalc += clock() - start;
931     if ((*param).VERBOSE) && (N > 0)
932     {
933         sprintf(text, "Number of examples processed: %d\n", numIter);
934         svmPrintString(text);
935     }
936     if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
937     {
938         sprintf(text, "Epoch %d/%d done.\n", epoch + 1, (*param).NUM_EPOCHS);
939         svmPrintString(text);
940     }
941     delete [] tempArray;
942     trainData->flushData();
943     if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
944         svmPrintString("\n");
945     if ((*param).VERBOSE)
946     {
947         sprintf(text, "Training completed in %5.3f seconds.\n\nNumber of budget maintenance steps:
948 %d\n", (double)timeCalc / (double)CLOCKS_PER_SEC, countDel);
949         svmPrintString(text);
950     }
951 }
952 }

```

6.3 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSV↵ M/src/budgetedSVM.h File Reference

Header file defining classes and functions used throughout the budgetedSVM toolbox.

Classes

- struct [parameters](#)
Structure holds the parameters of the implemented algorithms.
- class [budgetedData](#)
Class which handles manipulation of large data sets that cannot be fully loaded to memory (using a data structure similar to Matlab's sparse matrix structure).
- class [budgetedVector](#)
Class which handles high-dimensional vectors.
- class [budgetedModel](#)
Interface which defines methods to load model from and save model to text file.

Macros

- `#define` [INF](#) HUGE_VAL
Large (infinite) value, similar to Matlab's Inf.

Typedefs

- `typedef void(*` [funcPtr](#) `)(const char *text)`
Defines pointer to a function that prints information for a user, defined for more clear code.

Enumerations

- enum { **PEGASOS**, **AMM_BATCH**, **AMM_ONLINE**, **LLSVM**, **BSGD** }
Available large-scale, non-linear algorithms (note: unlike other algorithms, PEGASOS is a linear SVM solver).
- enum { **KERNEL_FUNC_GAUSSIAN**, **KERNEL_FUNC_EXPONENTIAL**, **KERNEL_FUNC_POLYNOMIAL**, **KERNEL_FUNC_LINEAR**, **KERNEL_FUNC_SIGMOID**, **KERNEL_FUNC_USER_DEFINED** }
Available kernel functions in kernel-based algorithms.
- enum { **LANDMARK_SAMPLE_RANDOM**, **LANDMARK_SAMPLE_KMEANS**, **LANDMARK_SAMPLE_KMEDOIDS** }
Available landmark sampling strategies in LLSVM.
- enum { **BUDGET_MAINTAIN_REMOVE**, **BUDGET_MAINTAIN_MERGE** }
Available budget maintenance strategies in BSGD.

Functions

- void [svmPrintString](#) (const char *text)
Prints string to the output.
- void [setPrintStringFunction](#) (funcPtr printFunc)
Modifies a callback that prints a string.
- void [svmPrintErrorString](#) (const char *text)
Prints error string to the output.
- void [setPrintErrorStringFunction](#) (funcPtr printFunc)
Modifies a callback that prints an error string.

- bool `fgetWord` (FILE *fHandle, char *str)
Reads one word string from an input file.
- bool `readableFileExists` (const char fileName[])
Checks if the file, identified by the input parameter, exists and is available for reading.
- void `parseInputPrompt` (int argc, char **argv, bool trainingPhase, char *inputFile, char *modelFile, char *outputFile, `parameters` *param)
Parses the user input from command prompt and modifies parameter settings as necessary, taken from LIBLINEAR implementation.
- void `printUsagePrompt` (bool trainingPhase, `parameters` *param)
Prints the instructions on how to use the software to standard output.

6.3.1 Detailed Description

Header file defining classes and functions used throughout the budgetedSVM toolbox.

6.3.2 Function Documentation

6.3.2.1 `fgetWord()`

```
bool fgetWord (
    FILE * fHandle,
    char * str )
```

Reads one word string from an input file.

Parameters

in	<i>fHandle</i>	Handle to an open file from which one word is read.
out	<i>str</i>	A character string that will hold the read word.

Returns

True if end-of-line or end-of-file encountered after reading a word string, otherwise false.

The function is similar to C++ functions `fgetc()` and `getline()`, only that it reads a single word from a text file. For the purposes of this project, a word is defined as a sequence of characters that does not contain a white-space character or new-line character ' '.

'.'. As a model in BudgetedSVM is stored in a text file where each line may corresponds to a single support vector, it is also useful to know if we reached the end of the line or the end of the file, which is indicated by the return value of the function.

Definition at line 37 of file `budgetedSVM.cpp`.

```
38 {
39     char temp;
40     unsigned char index = 0;
41     bool wordStarted = false;
42     while (1)//for (int i = 0; i < 20; i++)
43     {
44         temp = (char) fgetc(fHandle);
```

```

45
46     if (temp == EOF)
47     {
48         str[index++] = '\0';
49         return true;
50     }
51
52     switch (temp)
53     {
54         case ' ':
55             if (wordStarted)
56             {
57                 str[index++] = '\0';
58                 return false;
59             }
60             break;
61
62         case '\n':
63             str[index++] = '\0';
64             return true;
65             break;
66
67         default:
68             wordStarted = true;
69             str[index++] = temp;
70             break;
71     }
72 }
73 }

```

6.3.2.2 parseInputPrompt()

```

void parseInputPrompt (
    int argc,
    char ** argv,
    bool trainingPhase,
    char * inputFile,
    char * modelFile,
    char * outputFile,
    parameters * param )

```

Parses the user input from command prompt and modifies parameter settings as necessary, taken from LIBLINEAR implementation.

Parameters

in	<i>argc</i>	Argument count.
in	<i>argv</i>	Argument vector.
in	<i>trainingPhase</i>	True for training phase parsing, false for testing phase.
out	<i>inputFile</i>	Filename of input data file.
out	<i>modelFile</i>	Filename of model file.
out	<i>outputFile</i>	Filename of output file (only used during testing phase).
out	<i>param</i>	Parameter object modified by user input.

Definition at line 1303 of file budgetedSVM.cpp.

```

1304 {
1305     vector <char> option;
1306     vector <float> value;
1307     int i;
1308     FILE *pFile = NULL;
1309     char text[1024];
1310
1311     // parse options
1312     for (i = 1; i < argc; i++)

```

```

1313     {
1314         if (argv[i][0] != '-')
1315             break;
1316         ++i;
1317         option.push_back(argv[i - 1][1]);
1318         value.push_back((float) atof(argv[i]));
1319     }
1320
1321     if (trainingPhase)
1322     {
1323         if (i >= argc)
1324         {
1325             svmPrintErrorString("Error, input format not recognized. Run 'budgetedsvm-train' for
help.\n");
1326         }
1327
1328         pFile = fopen(argv[i], "r");
1329         if (pFile == NULL)
1330         {
1331             sprintf(text, "Can't open input file %s!\n", argv[i]);
1332             svmPrintErrorString(text);
1333         }
1334         else
1335         {
1336             fclose(pFile);
1337             strcpy(inputFile, argv[i]);
1338         }
1339
1340         // take model file if provided by a user
1341         if (i < argc - 1)
1342             strcpy(modelFile, argv[i + 1]);
1343         else
1344         {
1345             char *p = strrchr(argv[i], '/');
1346             if (p == NULL)
1347                 p = argv[i];
1348             else
1349                 ++p;
1350             sprintf(modelFile, "%s.model", p);
1351         }
1352
1353         // modify parameters
1354         for (unsigned int i = 0; i < option.size(); i++)
1355         {
1356             switch (option[i])
1357             {
1358                 case 'A':
1359                     (*param).ALGORITHM = (unsigned int) value[i];
1360                     if ((*param).ALGORITHM > 4)
1361                     {
1362                         sprintf(text, "Input parameter '-A %d' out of bounds!\nRun 'budgetedsvm-train'
for help.\n", (*param).ALGORITHM);
1363                         svmPrintErrorString(text);
1364                     }
1365                     break;
1366
1367                 case 'e':
1368                     (*param).NUM_EPOCHS = (unsigned int) value[i];
1369                     break;
1370
1371                 case 'D':
1372                     (*param).DIMENSION = (unsigned int) value[i];
1373                     break;
1374
1375                 case 's':
1376                     (*param).NUM_SUBEPOCHS = (unsigned int) value[i];
1377                     break;
1378
1379                 case 'k':
1380                     (*param).K_PARAM = (unsigned int) value[i];
1381                     break;
1382
1383                 case 'c':
1384                     (*param).C_PARAM = (double) value[i];
1385                     if ((*param).C_PARAM < 0.0)
1386                     {
1387                         sprintf(text, "Input parameter '-c' should be a non-negative real number!\nRun
'budgetedsvm-train' for help.\n");
1388                         svmPrintErrorString(text);
1389                     }
1390                     break;
1391
1392                 case 'L':
1393                     (*param).LAMBDA_PARAM = (double) value[i];
1394                     if ((*param).LAMBDA_PARAM <= 0.0)
1395                     {
1396                         sprintf(text, "Input parameter '-L' should be a positive real number!\nRun

```



```

    'budgetedsvm-train' for help.\n");
1397     svmPrintErrorString(text);
1398 }
1399 break;
1400
1401 case 'B':
1402     (*param).BUDGET_SIZE = (unsigned int) value[i];
1403     if ((*param).BUDGET_SIZE < 1)
1404     {
1405         sprintf(text, "Input parameter '-B' should be a positive integer!\nRun
'budgetedsvm-train' for help.\n");
1406         svmPrintErrorString(text);
1407     }
1408     break;
1409
1410 case 'g':
1411     (*param).KERNEL_GAMMA_PARAM = (double) value[i];
1412     if ((*param).KERNEL_GAMMA_PARAM <= 0.0)
1413     {
1414         sprintf(text, "Input parameter '-g' should be a positive real number!\nRun
'budgetedsvm-train' for help.\n");
1415         svmPrintErrorString(text);
1416     }
1417     break;
1418
1419 case 'd':
1420     (*param).KERNEL_DEGREE_PARAM = (double) value[i];
1421     if ((*param).KERNEL_DEGREE_PARAM <= 0.0)
1422     {
1423         sprintf(text, "Input parameter '-d' should be a positive real number!\nRun
'budgetedsvm-train' for help.\n");
1424         svmPrintErrorString(text);
1425     }
1426     break;
1427
1428 case 'i':
1429     (*param).KERNEL_COEF_PARAM = (double) value[i];
1430     break;
1431
1432 case 'K':
1433     (*param).KERNEL = (unsigned int) value[i];
1434     if ((*param).KERNEL > 5)
1435     {
1436         sprintf(text, "Input parameter '-K %d' out of bounds!\nRun 'budgetedsvm-train'
for help.\n", (*param).KERNEL);
1437         svmPrintErrorString(text);
1438     }
1439     break;
1440
1441 case 'm':
1442     (*param).MAINTENANCE_SAMPLING_STRATEGY = (unsigned int) value[i];
1443     break;
1444
1445 case 'b':
1446     (*param).BIAS_TERM = (double) value[i];
1447     break;
1448
1449 case 'v':
1450     (*param).VERBOSE = (value[i] != 0);
1451     break;
1452
1453 case 'z':
1454     (*param).CHUNK_SIZE = (unsigned int) value[i];
1455     if ((*param).CHUNK_SIZE < 1)
1456     {
1457         sprintf(text, "Input parameter '-z' should be a positive real number!\nRun
'budgetedsvm-train' for help.\n");
1458         svmPrintErrorString(text);
1459     }
1460     break;
1461
1462 case 'w':
1463     (*param).CHUNK_WEIGHT = (unsigned int) value[i];
1464     if ((*param).CHUNK_WEIGHT < 1)
1465     {
1466         sprintf(text, "Input parameter '-w' should be a positive real number!\nRun
'budgetedsvm-train' for help.\n");
1467         svmPrintErrorString(text);
1468     }
1469     break;
1470
1471 case 'S':
1472     (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
1473     break;
1474
1475 case 'r':
1476     (*param).RANDOMIZE = (value[i] != 0);

```

```

1477         break;
1478
1479         case 'C':
1480             (*param).CLONE_PROBABILITY = (double) value[i];
1481             if (((*param).CLONE_PROBABILITY < 0.0) || ((*param).CLONE_PROBABILITY > 1.0))
1482             {
1483                 sprintf(text, "Input parameter '-C' should be a real number between 0 and
1484 1!\nRun 'budgetedsvm-train()' for help.\n");
1485                 svmPrintErrorString(text);
1486             }
1487             break;
1488
1489         case 'y':
1490             (*param).CLONE_PROBABILITY_DECAY = (double) value[i];
1491             if (((*param).CLONE_PROBABILITY_DECAY < 0.0) || ((*param).CLONE_PROBABILITY_DECAY >
1492 1.0))
1493             {
1494                 sprintf(text, "Input parameter '-y' should be a real number between 0 and
1495 1!\nRun 'budgetedsvm-train()' for help.\n");
1496                 svmPrintErrorString(text);
1497             }
1498             break;
1499
1500         default:
1501             sprintf(text, "Error, unknown input parameter '-%c'!\nRun 'budgetedsvm-train' for
1502 help.\n", option[i]);
1503             svmPrintErrorString(text);
1504             break;
1505     }
1506 }
1507
1508 // for BSGD, when we use merging budget maintenance strategy then only Gaussian kernel can be
1509 used,
1510 // due to the nature of merging; here check if user specified some other kernel while merging
1511 if (((*param).ALGORITHM == BSGD) && ((*param).KERNEL != KERNEL_FUNC_GAUSSIAN) &&
1512 ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE))
1513 {
1514     svmPrintString("Warning, BSGD with merging strategy can only use Gaussian kernel!\nKernel
1515 function switched to Gaussian.\n");
1516     (*param).KERNEL = KERNEL_FUNC_GAUSSIAN;
1517 }
1518
1519 // signal error if a user wants to use RBF kernel, but didn't specify either data dimension or
1520 kernel width
1521 if ((((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD)) && ((*param).KERNEL ==
1522 KERNEL_FUNC_GAUSSIAN) || ((*param).KERNEL == KERNEL_FUNC_EXPONENTIAL))
1523 {
1524     if (((*param).KERNEL_GAMMA_PARAM == 0.0) && ((*param).DIMENSION == 0))
1525     {
1526         // this means that both RBF kernel width and dimension were not set by the user in the
1527         input string to the toolbox
1528         // since in this case the default value of RBF kernel is 1/dimensionality, report
1529         error to the user
1530         svmPrintErrorString("Error, RBF kernel in use, please set either kernel width or
1531 dimensionality!\nRun 'budgetedsvm-train' for help.\n");
1532     }
1533 }
1534
1535 // check the MAINTENANCE_SAMPLING_STRATEGY validity
1536 if ((*param).ALGORITHM == LLSVM)
1537 {
1538     if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 2)
1539     {
1540         // 0 - random removal, 1 - k-means, 2 - k-medoids
1541         sprintf(text, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm-train' for
1542 help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
1543         svmPrintErrorString(text);
1544     }
1545 }
1546
1547 else if ((*param).ALGORITHM == BSGD)
1548 {
1549     if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 1)
1550     {
1551         // 0 - smallest removal, 1 - merging
1552         sprintf(text, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm-train' for
1553 help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
1554         svmPrintErrorString(text);
1555     }
1556 }
1557 }
1558
1559 // shut down printing to screen if user specified so
1560 if (!(*param).VERBOSE)
1561     setPrintStringFunction(NULL);
1562
1563 // no bias term for LLSVM and BSGD functions
1564 if (((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD))
1565 {

```

```

1550         (*param).BIAS_TERM = 0.0;
1551     }
1552
1553     if ((*param).VERBOSE)
1554     {
1555         svmPrintString("\n*** Training started with the following parameters:\n");
1556         switch ((*param).ALGORITHM)
1557         {
1558             case PEGASOS:
1559                 svmPrintString("Algorithm \t\t\t: Pegasos\n");
1560                 break;
1561             case AMM_ONLINE:
1562                 svmPrintString("Algorithm \t\t\t: AMM online\n");
1563                 break;
1564             case AMM_BATCH:
1565                 svmPrintString("Algorithm \t\t\t: AMM batch\n");
1566                 break;
1567             case BSGD:
1568                 svmPrintString("Algorithm \t\t\t: BSGD\n");
1569                 break;
1570             case LLSVM:
1571                 svmPrintString("Algorithm \t\t\t: LLSVM\n");
1572                 break;
1573         }
1574
1575         if (((*param).ALGORITHM == PEGASOS) || ((*param).ALGORITHM == AMM_BATCH) ||
            ((*param).ALGORITHM == AMM_ONLINE))
1576         {
1577             sprintf(text, "Lambda parameter\t\t: %f\n", (*param).LAMBDA_PARAM);
1578             svmPrintString(text);
1579             sprintf(text, "Bias term \t\t\t: %f\n", (*param).BIAS_TERM);
1580             svmPrintString(text);
1581             if ((*param).ALGORITHM != PEGASOS)
1582             {
1583                 sprintf(text, "Pruning frequency k \t\t: %d\n", (*param).K_PARAM);
1584                 svmPrintString(text);
1585                 sprintf(text, "Pruning parameter c \t\t: %.2f\n", (*param).C_PARAM);
1586                 svmPrintString(text);
1587                 sprintf(text, "Max num. of weights per class \t: %d\n", (*param).BUDGET_SIZE);
1588                 svmPrintString(text);
1589                 sprintf(text, "Number of epochs \t\t: %d\n", (*param).NUM_EPOCHS);
1590                 svmPrintString(text);
1591             }
1592             else
1593                 svmPrintString("\n");
1594         }
1595         else if (((*param).ALGORITHM == BSGD) || ((*param).ALGORITHM == LLSVM))
1596         {
1597             if ((*param).ALGORITHM == BSGD)
1598             {
1599                 sprintf(text, "Number of epochs \t\t: %d\n", (*param).NUM_EPOCHS);
1600                 svmPrintString(text);
1601                 if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_REMOVE)
1602                     svmPrintString("Maintenance strategy \t\t: 0 (smallest removal)\n");
1603                 else if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE)
1604                     svmPrintString("Maintenance strategy \t\t: 1 (merging)\n");
1605                 else
1606                     svmPrintErrorString("Error, unknown budget maintenance set. Run
'budgetedsvm-train' for help.\n");
1607
1608                 svmPrintString(text);
1609                 sprintf(text, "Size of the budget \t\t: %d\n", (*param).BUDGET_SIZE);
1610                 svmPrintString(text);
1611             }
1612             else if ((*param).ALGORITHM == LLSVM)
1613             {
1614                 switch ((*param).MAINTENANCE_SAMPLING_STRATEGY)
1615                 {
1616                     case LANDMARK_SAMPLE_RANDOM:
1617                         svmPrintString("Landmark sampling \t\t: 0 (random sampling)\n");
1618                         break;
1619                     case LANDMARK_SAMPLE_KMEANS:
1620                         svmPrintString("Landmark sampling \t\t: k-means initialization\n");
1621                         break;
1622                     case LANDMARK_SAMPLE_KMEDOIDS:
1623                         svmPrintString("Landmark sampling \t\t: 1 (k-medoids initialization)\n");
1624                         break;
1625                     default:
1626                         svmPrintErrorString("Error, unknown landmark sampling set. Run
'budgetedsvm-train' for help.\n");
1627                         break;
1628                 }
1629                 sprintf(text, "Number of landmark points \t: %d\n", (*param).BUDGET_SIZE);
1630                 svmPrintString(text);
1631             }
1632
1633             // now print the common parameters

```

```

1634         sprintf(text, "Lambda regularization param. \t: %f\n", (*param).LAMBDA_PARAM);
1635         svmPrintString(text);
1636         switch ((*param).KERNEL)
1637         {
1638             case KERNEL_FUNC_GAUSSIAN:
1639                 svmPrintString("Gaussian kernel used \t\t:  $K(x, y) = \exp(-0.5 * \gamma * ||x - y||^2)$ \n");
1640                 if ((*param).KERNEL_GAMMA_PARAM != 0.0)
1641                 {
1642                     sprintf(text, "Kernel width gamma \t\t: %f\n",
1643                         (*param).KERNEL_GAMMA_PARAM);
1644                     svmPrintString(text);
1645                 }
1646                 else
1647                     svmPrintString("Kernel width gamma \t\t: 1 / DIMENSIONALITY\n");
1648                 break;
1649             case KERNEL_FUNC_EXPONENTIAL:
1650                 svmPrintString("Exponential kernel used \t:  $K(x, y) = \exp(-0.5 * \gamma * ||x - y||)$ \n");
1651                 if ((*param).KERNEL_GAMMA_PARAM != 0.0)
1652                 {
1653                     sprintf(text, "Kernel width gamma \t\t: %f\n",
1654                         (*param).KERNEL_GAMMA_PARAM);
1655                     svmPrintString(text);
1656                 }
1657                 else
1658                     svmPrintString("Kernel width gamma \t\t: 1 / DIMENSIONALITY\n");
1659                 break;
1660             case KERNEL_FUNC_POLYNOMIAL:
1661                 sprintf(text, "Polynomial kernel used \t\t:  $K(x, y) = (x^T * y +$ 
1662                      $0.2f)^{0.2f}$ \n", (*param).KERNEL_COEF_PARAM, (*param).KERNEL_DEGREE_PARAM);
1663                 svmPrintString(text);
1664                 break;
1665             case KERNEL_FUNC_SIGMOID:
1666                 sprintf(text, "Sigmoid kernel used \t\t:  $K(x, y) = \tanh(0.2f * x^T * y +$ 
1667                      $0.2f)$ \n", (*param).KERNEL_DEGREE_PARAM, (*param).KERNEL_COEF_PARAM);
1668                 svmPrintString(text);
1669                 break;
1670             case KERNEL_FUNC_LINEAR:
1671                 svmPrintString("Linear kernel used \t\t:  $K(x, y) = (x^T * y)$ \n");
1672                 break;
1673             case KERNEL_FUNC_USER_DEFINED:
1674                 svmPrintString("User-defined kernel function used.\n");
1675                 break;
1676         }
1677     }
1678 }
1679 }
1680
1681 // increase dimensionality if bias term included
1682 if ((*param).BIAS_TERM != 0.0)
1683     (*param).DIMENSION++;
1684
1685 // set gamma to default value of inverse dimensionality if not specified by a user
1686 if ((*param).KERNEL_GAMMA_PARAM == 0.0)
1687     (*param).KERNEL_GAMMA_PARAM = 1.0 / (*param).DIMENSION;
1688 }
1689 else
1690 {
1691     if (i >= argc - 2)
1692     {
1693         svmPrintErrorString("Error, input format not recognized. Run 'budgetedsvm-predict' for
1694             help.\n");
1695     }
1696     pFile = fopen(argv[i], "r");
1697     if (pFile == NULL)
1698     {
1699         sprintf(text, "Can't open input file %s!\n", argv[i]);
1700         svmPrintErrorString(text);
1701     }
1702     else
1703     {
1704         fclose(pFile);
1705         strcpy(inputFile, argv[i]);
1706     }
1707
1708     pFile = fopen(argv[i + 1], "r");
1709     if (pFile == NULL)
1710     {
1711         sprintf(text, "Can't open model file %s!\n", argv[i + 1]);
1712         svmPrintErrorString(text);
1713     }

```

```

1714         else
1715         {
1716             fclose(pFile);
1717             strcpy(modelFile, argv[i + 1]);
1718         }
1719
1720         pFile = fopen(argv[i + 2], "w");
1721         if (pFile == NULL)
1722         {
1723             sprintf(text, "Can't create output file %s!\n", argv[i + 2]);
1724             svmPrintErrorString(text);
1725         }
1726         else
1727         {
1728             fclose(pFile);
1729             strcpy(outputFile, argv[i + 2]);
1730         }
1731
1732         // modify parameters
1733         for (unsigned int i = 0; i < option.size(); i++)
1734         {
1735             switch (option[i])
1736             {
1737                 case 'v':
1738                     (*param).VERBOSE = (value[i] != 0);
1739                     break;
1740
1741                 case 'z':
1742                     (*param).CHUNK_SIZE = (unsigned int) value[i];
1743                     if ((*param).CHUNK_SIZE < 1)
1744                     {
1745                         sprintf(text, "Input parameter '-z' should be a positive real number!\nRun
'budgetedsvm-predict' for help.\n");
1746                         svmPrintErrorString(text);
1747                     }
1748                     break;
1749
1750                 case 'w':
1751                     (*param).CHUNK_WEIGHT = (unsigned int) value[i];
1752                     if ((*param).CHUNK_WEIGHT < 1)
1753                     {
1754                         sprintf(text, "Input parameter '-w' should be a positive real number!\nRun
'budgetedsvm-predict' for help.\n");
1755                         svmPrintErrorString(text);
1756                     }
1757                     break;
1758
1759                 case 'S':
1760                     (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
1761                     break;
1762
1763                 case 'o':
1764                     (*param).OUTPUT_SCORES = (value[i] != 0);
1765                     break;
1766
1767                 default:
1768                     sprintf(text, "Error, unknown input parameter '-%c'!\nRun 'budgetedsvm-predict' for
help.\n", option[i]);
1769                     svmPrintErrorString(text);
1770                     break;
1771             }
1772         }
1773
1774         // shut down printing to screen if user specified so
1775         if (!(*param).VERBOSE)
1776             setPrintStringFunction(NULL);
1777     }
1778 }

```

6.3.2.3 printUsagePrompt()

```

void printUsagePrompt (
    bool trainingPhase,
    parameters * param )

```

Prints the instructions on how to use the software to standard output.

Parameters

in	<i>trainingPhase</i>	Indicator if training or testing phase instructions.
in	<i>param</i>	Parameter object modified by user input.

Definition at line 1187 of file budgetedSVM.cpp.

```

1188 {
1189     char text[256];
1190     if (trainingPhase)
1191     {
1192         svmPrintString("\n Usage:\n");
1193         svmPrintString(" budgetedsvm-train [options] train_file [model_file]\n\n");
1194         svmPrintString(" Inputs:\n");
1195         svmPrintString(" options\t- parameters of the model\n");
1196         svmPrintString(" train_file\t- url of training file in LIBSVM format\n");
1197         svmPrintString(" model_file\t- file that will hold a learned model\n");
1198         svmPrintString(" -----");
1199         svmPrintString(" Options are specified in the following format:\n");
1200         svmPrintString(" '-OPTION1 VALUE1 -OPTION2 VALUE2 ...'\n\n");
1201         svmPrintString(" Following options are available; affected algorithm and default values
are\n");
1202         svmPrintString(" given in parentheses (algorithm not specified if option affects all):\n\n");
1203         sprintf(text, " A - algorithm, which large-scale SVM approximation to use (%d):\n",
(*param).ALGORITHM);
1204         svmPrintString(text);
1205         svmPrintString("          0 - Pegasos\n");
1206         svmPrintString("          1 - AMM batch\n");
1207         svmPrintString("          2 - AMM online\n");
1208         svmPrintString("          3 - LLSVM\n");
1209         svmPrintString("          4 - BSGD\n");
1210         svmPrintString(" D - dimensionality (faster loading if set, if omitted inferred from the
data)\n");
1211         svmPrintString(" B - limit on the number of weights per class in AMM, OR\n");
1212         sprintf(text, "          total SV set budget in BSGD, OR number of landmark points in LLSVM
(%d)\n", (*param).BUDGET_SIZE);
1213         svmPrintString(text);
1214         sprintf(text, " L - lambda regularization parameter; high value -> less complex model
(%.5f)\n", (*param).LAMBDA_PARAM);
1215         svmPrintString(text);
1216         sprintf(text, " b - bias term, if 0 no bias added (%.1f)\n", (*param).BIAS_TERM);
1217         svmPrintString(text);
1218         sprintf(text, " e - number of training epochs (AMM, BSGD; %d)\n", (*param).NUM_EPOCHS);
1219         svmPrintString(text);
1220         sprintf(text, " s - number of subepochs (AMM batch; %d)\n", (*param).NUM_SUBEPOCHS);
1221         svmPrintString(text);
1222         sprintf(text, " k - pruning frequency, after how many examples is pruning done (AMM; %d)\n",
(*param).K_PARAM);
1223         svmPrintString(text);
1224         sprintf(text, " c - pruning threshold; high value -> less complex model (AMM; %.2f)\n",
(*param).C_PARAM);
1225         svmPrintString(text);
1226         svmPrintString(" K - kernel function (0 - RBF; 1 - exponential, 2 - polynomial; 3 - linear,
\n");
1227         sprintf(text, "          4 - sigmoid; 5 - user-defined) (LLSVM, BSGD; %d)\n", (*param).KERNEL);
1228         svmPrintString(text);
1229         sprintf(text, " g - RBF or exponential kernel width gamma (LLSVM, BSGD; 1/DIMENSIONALITY)\n");
1230         svmPrintString(text);
1231         sprintf(text, " d - polynomial kernel degree or sigmoid kernel slope (LLSVM, BSGD; %.2f)\n",
(*param).KERNEL_DEGREE_PARAM);
1232         svmPrintString(text);
1233         sprintf(text, " i - polynomial or sigmoid kernel intercept (LLSVM, BSGD; %.2f)\n",
(*param).KERNEL_COEF_PARAM);
1234         svmPrintString(text);
1235         svmPrintString(" m - budget maintenance in BSGD (0 - removal; 1 - merging, uses Gaussian
kernel), OR\n");
1236         sprintf(text, "          landmark selection in LLSVM (0 - random; 1 - k-means; 2 - k-medoids)
(%d)\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
1237         svmPrintString(text);
1238
1239         sprintf(text, " C - clone probability when misclassification occurs in AMM (%.2f)\n",
(*param).CLONE_PROBABILITY);
1240         svmPrintString(text);
1241         sprintf(text, " y - clone probability decay when weight cloning occurs in AMM (%.2f)\n\n",
(*param).CLONE_PROBABILITY_DECAY);
1242         svmPrintString(text);
1243
1244         svmPrintString(" z - training and test file are loaded in chunks so that the algorithms
can\n");
1245         svmPrintString("          handle budget files on weaker computers; z specifies number of
examples\n");
1246         sprintf(text, "          loaded in a single chunk of data (%d)\n", (*param).CHUNK_SIZE);
1247         svmPrintString(text);
1248         svmPrintString(" w - model weights are split in chunks, so that the algorithm can handle\n");

```

```

1249     svmPrintString("          highly dimensional data on weaker computers; w specifies number of\n");
1250     sprintf(text, "          dimensions stored in one chunk (%d)\n", (*param).CHUNK_WEIGHT);
1251     svmPrintString(text);
1252     svmPrintString(" S - if set to 1 data is assumed sparse, if 0 data assumed non-sparse; used
to\n");
1253     svmPrintString("          speed up kernel computations (default is 1 when percentage of
non-zero\n");
1254     svmPrintString("          features is less than 5%, and 0 when percentage is larger than 5%)\n");
1255     sprintf(text, " r - randomize the algorithms; 1 to randomize, 0 not to randomize (%d)\n",
(*param).RANDOMIZE);
1256     svmPrintString(text);
1257     sprintf(text, " v - verbose output; 1 to show the algorithm steps, 0 for quiet mode (%d)\n\n",
(*param).VERBOSE);
1258     svmPrintString(text);
1259 }
1260 else
1261 {
1262     svmPrintString("\n Usage:\n");
1263     svmPrintString(" budgetedsvm-predict [options] test_file model_file output_file\n\n");
1264     svmPrintString(" Inputs:\n");
1265     svmPrintString(" options\t- parameters of the model\n");
1266     svmPrintString(" test_file\t- url of test file in LIBSVM format\n");
1267     svmPrintString(" model_file\t- file that holds a learned model\n");
1268     svmPrintString(" output_file\t- url of file where output will be written\n");
1269     svmPrintString(" -----\n");
1270     svmPrintString(" Options are specified in the following format:\n");
1271     svmPrintString(" '-OPTION1 VALUE1 -OPTION2 VALUE2 ...'\n\n");
1272     svmPrintString(" The following options are available (default values in parentheses):\n\n");
1273
1274     svmPrintString(" z - the training and test file are loaded in chunks so that the algorithm
can\n");
1275     svmPrintString("          handle budget files on weaker computers; z specifies number of
examples\n");
1276     sprintf(text, "          loaded in a single chunk of data (%d)\n", (*param).CHUNK_SIZE);
1277     svmPrintString(text);
1278     svmPrintString(" w - the model weight is split in parts, so that the algorithm can handle\n");
1279     svmPrintString("          highly dimensional data on weaker computers; w specifies number of\n");
1280     sprintf(text, "          dimensions stored in one chunk (%d)\n", (*param).CHUNK_WEIGHT);
1281     svmPrintString(text);
1282     svmPrintString(" S - if set to 1 data is assumed sparse, if 0 data assumed non-sparse, used
to\n");
1283     svmPrintString("          speed up kernel computations (default is 1 when percentage of
non-zero\n");
1284     svmPrintString("          features is less than 5%, and 0 when percentage is larger than 5%)\n");
1285     svmPrintString(" o - if set to 1, the output file will contain not only the class
predictions,\n");
1286     sprintf(text, "          but also tab-delimited scores of the winning class (%d)\n",
(*param).OUTPUT_SCORES);
1287     svmPrintString(text);
1288     sprintf(text, " v - verbose output; 1 to show algorithm steps, 0 for quiet mode (%d)\n\n",
(*param).VERBOSE);
1289     svmPrintString(text);
1290 }
1291 }

```

6.3.2.4 readableFileExists()

```

bool readableFileExists (
    const char fileName[] )

```

Checks if the file, identified by the input parameter, exists and is available for reading.

Parameters

in	<i>fileName</i>	Handle to an open file from which one word is read.
----	-----------------	---

Returns

True if the file exists and is available for reading, otherwise false.

Definition at line 162 of file budgetedSVM.cpp.

```

163 {
164     FILE *pFile = NULL;
165     if (fileName)
166     {
167         pFile = fopen(fileName, "r");
168         if (pFile != NULL)
169         {
170             fclose(pFile);
171             return true;
172         }
173     }
174     return false;
175 }

```

6.3.2.5 setPrintErrorStringFunction()

```

void setPrintErrorStringFunction (
    funcPtr printFunc )

```

Modifies a callback that prints an error string.

Parameters

in	<i>printFunc</i>	New text-printing function.
----	------------------	-----------------------------

This function is used to modify the function that is used to print to error output. After calling this function, which modifies the callback function for printing error string, the text is printed simply by invoking [svmPrintErrorString](#).

See also

[funcPtr](#)

Definition at line 152 of file budgetedSVM.cpp.

```

153 {
154     svmPrintErrorStringStatic = (printFunc == NULL) ? &printErrorDefault : printFunc;
155 }

```

6.3.2.6 setPrintStringFunction()

```

void setPrintStringFunction (
    funcPtr printFunc )

```

Modifies a callback that prints a string.

Parameters

in	<i>printFunc</i>	New text-printing function.
----	------------------	-----------------------------

This function is used to modify the function that is used to print to standard output. After calling this function, which modifies the callback function for printing, the text is printed simply by invoking [svmPrintString](#).

See also

[funcPtr](#)

Definition at line 126 of file budgetedSVM.cpp.

```
127 {  
128     if (printFunc == NULL)  
129         svmPrintStringStatic = &printNull;  
130     else  
131         svmPrintStringStatic = printFunc;  
132 }
```

6.3.2.7 svmPrintErrorString()

```
void svmPrintErrorString (  
    const char * text )
```

Prints error string to the output.

Parameters

in	<i>text</i>	Text to be printed.
----	-------------	---------------------

Prints error string to the output. Exactly to which output should be specified by [setPrintErrorStringFunction](#), which modifies the callback that is invoked for printing. This is convenient when an error is detected and, prior to printing appropriate message to a user, we want to exit the program. For example on how to set the printing function in Matlab environment, see the implementation of [parseInputMatlab](#).

Definition at line 140 of file budgetedSVM.cpp.

```
141 {  
142     svmPrintErrorStringStatic(text);  
143 }
```

6.3.2.8 svmPrintString()

```
void svmPrintString (  
    const char * text )
```

Prints string to the output.

Parameters

in	<i>text</i>	Text to be printed.
----	-------------	---------------------

Prints string to the output. Exactly to which output should be specified by [setPrintStringFunction](#), which modifies the callback that is invoked for printing. This is convenient when simple printf() can not be used, for example if we want to print to Matlab prompt. For example on how to set the printing function in Matlab environment, see the implementation of [parseInputMatlab](#).

Definition at line 114 of file budgetedSVM.cpp.

```
115 {  
116     svmPrintStringStatic(text);  
117 }
```

6.4 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/llsvm.h File Reference

Defines classes and functions used for training and testing of LLSVM algorithm.

Classes

- class [budgetedVectorLLSVM](#)
Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for LLSVM algorithm.
- class [budgetedModelLLSVM](#)
Class which holds the LLSVM model, and implements methods to load LLSVM model from and save LLSVM model to text file.

Functions

- void [trainLLSVM](#) ([budgetedData](#) *trainData, [parameters](#) *param, [budgetedModelLLSVM](#) *model)
Train LLSVM online.
- float [predictLLSVM](#) ([budgetedData](#) *testData, [parameters](#) *param, [budgetedModelLLSVM](#) *model, vector< int > *labels=NULL, vector< float > *scores=NULL)
Given an LLSVM model, predict the labels of testing data.

6.4.1 Detailed Description

Defines classes and functions used for training and testing of LLSVM algorithm.

6.4.2 Function Documentation

6.4.2.1 predictLLSVM()

```
float predictLLSVM (
    budgetedData * testData,
    parameters * param,
    budgetedModelLLSVM * model,
    vector< int > * labels,
    vector< float > * scores )
```

Given an LLSVM model, predict the labels of testing data.

Parameters

in	<i>testData</i>	Input test data.
in	<i>param</i>	The parameters of the algorithm.
in	<i>model</i>	Trained LLSVM model.
out	<i>labels</i>	Vector of predicted labels.
out	<i>scores</i>	Vector of scores of the winning labels.

Returns

Testing set error rate.

Given the learned BSGD model, the function computes the predictions on the testing data, outputting the predicted labels and the error rate.

Definition at line 362 of file llsvm.cpp.

```

363 {
364     // to train linear kernel we need -1 and +1 labels, but a user can give us any labels, e.g., 0/1
    labels; therefore
365     // here we set the default labels, such that the first user-provided label is renamed as -1, and
    the second +1
366     int defaultLabels[2] = {-1, 1};
367
368     unsigned long N, err = 0, total = 0, timeCalc = 0, start;
369     bool stillChunksLeft = true;
370     char text[256];
371     VectorXd v((*param).BUDGET_SIZE), temp((*param).BUDGET_SIZE);
372     budgetedVectorLLSVM *currentData = NULL;
373     long double tempSqrNorm;
374
375     if ((*param).VERBOSE)
376         svmPrintString("Computing lower-dimensional representation and predicting labels ...\n");
377
378     while (stillChunksLeft)
379     {
380         stillChunksLeft = testData->readChunk((*param).CHUNK_SIZE);
381         (*param).updateVerySparseDataParameter(testData->getSparsity());
382
383         N = testData->N;
384         total += N;
385         start = clock();
386
387         // calculate E, kernel between testing points and landmark points
388         VectorXd predictions(N);
389         for (unsigned int i = 0; i < N; i++)
390         {
391             if ((*param).VERY_SPARSE_DATA)
392             {
393                 // since we are computing kernels using vectors directly from the budgetedData, we need
    square norm of the vector to speed-up
394                 // computations, here we compute it just once; no need to do it in non-sparse case,
    since this norm can be retrieved directly
395                 // from budgetedVector
396
397                 tempSqrNorm = testData->getVectorSqrL2Norm(i, param);
398                 for (unsigned int j = 0; j < (*param).BUDGET_SIZE; j++)
399                 {
400                     v(j) = (double) (*model->modelLLSVMlandmarks)[j]->computeKernel(i, testData, param,
    tempSqrNorm);
401                 }
402             }
403             else
404             {
405                 // first create the budgetedVector using the vector from budgetedData, to be used in
    gaussianKernel() method below
406                 currentData = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
407                 currentData->createVectorUsingDataPoint(testData, i, param);
408
409                 for (unsigned int j = 0; j < (*param).BUDGET_SIZE; j++)
410                 {
411                     v(j) = (double) (*model->modelLLSVMlandmarks)[j]->computeKernel(currentData,
    param);
412                 }
413                 delete currentData;
414                 currentData = NULL;
415             }
416
417             temp = v.transpose() * (*model).modelLLSVMmatrixW;
418             predictions(i) = temp.dot((*model).modelLLSVMweightVector);
419         }
420
421         for (unsigned int i = 0; i < N; i++)
422         {
423             if ((predictions(i) > 0.0) != (defaultLabels[testData->al[i]] > 0))
424                 err++;
425
426             // save predicted label, will be sent to output ...
427             if (labels)
428                 (*labels).push_back((int) (testData->yLabels)[(predictions(i) > 0)]);
429             // ... and the scores
430             if (scores)
431                 (*scores).push_back((float) abs(predictions(i)));

```

```

432     }
433
434     timeCalc += clock() - start;
435
436     if ((*param).VERBOSE) && (N > 0))
437     {
438         sprintf(text, "Number of examples processed: %ld\n", total);
439         svmPrintString(text);
440     }
441 }
442
443 if ((*param).VERBOSE)
444 {
445     sprintf(text, "*** Testing completed in %5.3f seconds\n*** Testing error rate: %3.2f
percent\n\n", (double)timeCalc / (double)CLOCKS_PER_SEC, 100.0 * (float) err / (float) total);
446     svmPrintString(text);
447 }
448
449 return (float) (100.0 * (float) err / (float) total);
450 }

```

6.4.2.2 trainLLSVM()

```

void trainLLSVM (
    budgetedData * trainData,
    parameters * param,
    budgetedModelLLSVM * model )

```

Train LLSVM online.

Parameters

in	<i>trainData</i>	Input training data.
in	<i>param</i>	The parameters of the algorithm.
in, out	<i>model</i>	Initial LLSVM model.

The function trains LLSVM model, given input data, the initial model (most often zero-weight model), and the parameters of the model.

Definition at line 556 of file llsvm.cpp.

```

557 {
558     unsigned long timeCalc = 0, start;
559     unsigned int i, j, total = 0, N, temp;
560     bool stillChunksLeft = true, firstChunk = true;
561     long double tempSqrNorm;
562     char text[256];
563     budgetedVectorLLSVM *currentData = NULL;
564
565     // W matrix for Nystrom method, here employ Eigen library since we need complex matrix operations
566     (*model).modelLLSVMmatrixW = MatrixXd::Zero((*param).BUDGET_SIZE, (*param).BUDGET_SIZE);
567
568     // initialize weight (i.e., hyperplane) in the projected space to zero-vector
569     (*model).modelLLSVMweightVector = VectorXd::Zero((*param).BUDGET_SIZE);
570
571     // commence with LLSVM training procedure
572     stillChunksLeft = true;
573     while (stillChunksLeft)
574     {
575         stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);
576
577         // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
578         // (of course, in the case of AMM, speeds up linear kernel computation)
579         (*param).updateVerySparseDataParameter(trainData->getSparsity());
580
581         // compute observed data dimensionality, where we also account for possible bias term, and check
582         // we need to expand the current model weights if some new data dimensions were found during
583         loading
584         temp = trainData->getDataDimensionality() + (int) (param->BIAS_TERM != 0.0);

```

```

584     if ((*param).DIMENSION < temp)
585     {
586         /*sprintf(text, "Extending the model, current: %d\found: %d\n", (*param).DIMENSION, temp);
587         svmPrintString(text);*/
588         (*model).extendDimensionalityOfModel(temp, param);
589
590         // update the dimensionality
591         (*param).DIMENSION = temp;
592     }
593
594     if (trainData->yLabels.size() != 2)
595     {
596         sprintf(text, "LLSVM is a binary classifier, but %d class(es) detected!\n", (int)
trainData->yLabels.size());
597         svmPrintErrorString(text);
598     }
599
600     N = trainData->N;
601     total += N;
602     start = clock();
603
604     // if we just started training initialize the landmark points
605     if (firstChunk)
606     {
607         if (N < (*param).BUDGET_SIZE)
608         {
609             trainData->flushData();
610             svmPrintErrorString("Number of landmark points larger than size of the loaded
chunk!\n");
611         }
612         firstChunk = false;
613
614         // select landmark points
615         selectLandmarkPoints(trainData, param, model);
616
617         if ((*param).VERBOSE)
618             svmPrintString("Computing the mapping function ...\n");
619
620         // compute the W matrix, done just once per training
621         for (i = 0; i < (*param).BUDGET_SIZE; i++)
622         {
623             for (j = 0; j < (*param).BUDGET_SIZE; j++)
624             {
625                 if (i <= j)
626                 {
627                     (*model).modelLLSVMmatrixW(i, j) = (double)
(*model->modelLLSVMlandmarks)[i]->computeKernel((*model->modelLLSVMlandmarks)[j], param);
628                 }
629                 else
630                 {
631                     (*model).modelLLSVMmatrixW(i, j) = (*model).modelLLSVMmatrixW(j, i);
632                 }
633             }
634         }
635
636         // finally, compute K_zz = W^(-0.5), initialization is complete
637         invSquareRoot((*model).modelLLSVMmatrixW);
638     }
639
640     // done with initialization phase, next we compute the mapping in the new space and solve linear
SVM
641
642     if ((*param).VERBOSE)
643         svmPrintString("Computing mapping of the training data ...\n");
644
645     // here compute kernel matrix E between input data and landmark points
646     MatrixXd E = MatrixXd::Zero(N, (*param).BUDGET_SIZE);
647     for (i = 0; i < N; i++)
648     {
649         if ((*param).VERY_SPARSE_DATA)
650         {
651             // since we are computing kernels using vectors directly from the budgetedData, we need
square norm of the vector to speed-up
652             // computations, here we compute it just once; no need to do it in non-sparse case,
since this norm can be retrieved directly
653             // from budgetedVector
654             tempSqrNorm = trainData->getVectorSqrL2Norm(i, param);
655             for (j = 0; j < (*param).BUDGET_SIZE; j++)
656             {
657                 E(i, j) = (double) (*model->modelLLSVMlandmarks)[j]->computeKernel(i, trainData,
param, tempSqrNorm);
658             }
659         }
660         else
661         {
662             // first create the budgetedVector using the vector from budgetedData, to be used in
gaussianKernel() method below

```

```

663         currentData = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
664         currentData->createVectorUsingDataPoint(trainData, i, param);
665
666         for (j = 0; j < (*param).BUDGET_SIZE; j++)
667         {
668             E(i, j) = (double) (*model->modelLLSVMlandmarks) [j]->computeKernel(currentData,
param);
669         }
670         delete currentData;
671         currentData = NULL;
672     }
673 }
674
675 // compute new representation of data set which will be used to train SVM
676 E = E * (*model).modelLLSVMmatrixW;
677
678 if ((*param).VERBOSE)
679     svmPrintString("Training linear SVM ...\n");
680
681 // now we can move on to training SVM using data in a new space
682 liblinear_Solve_l2r_l1(E, trainData->al, (*model).modelLLSVMweightVector, param,
&(trainData->yLabels));
683 timeCalc += clock() - start;
684
685 if ((*param).VERBOSE) && (N > 0))
686 {
687     sprintf(text, "Number of examples processed: %d\n", total);
688     svmPrintString(text);
689 }
690 }
691 // training done, get rid of training data
692 trainData->flushData();
693
694 //timeCalc += clock() - startTotal;
695 if ((*param).VERBOSE)
696 {
697     sprintf(text, "*** Training completed in %5.3f seconds.\n\n", (double)timeCalc /
(double)CLOCKS_PER_SEC);
698     svmPrintString(text);
699 }
700 }

```

6.5 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.h File Reference

Defines classes and functions used for training and testing of large-scale multi-hyperplane algorithms (AMM batch, AMM online, and Pegasos).

Classes

- class [budgetedVectorAMM](#)

Class which holds sparse vector, which is split into a number of arrays to trade-off between speed of access and memory usage of sparse data, with added methods for AMM algorithms.

- class [budgetedModelAMM](#)

Class which holds the AMM model, and implements methods to load AMM model from and save AMM model to text file.

Typedefs

- typedef vector< [budgetedVectorAMM](#) * > [vectorOfBudgetVectors](#)

A vector of vectors, implements the weight matrix of AMM algorithms as jagged array.

Functions

- void `trainPegasos` (`budgetedData` *trainData, `parameters` *param, `budgetedModelAMM` *model)
Train Pegasos.
- void `trainAMMonline` (`budgetedData` *trainData, `parameters` *param, `budgetedModelAMM` *model)
Train AMM online.
- void `trainAMMbatch` (`budgetedData` *trainData, `parameters` *param, `budgetedModelAMM` *model)
Train AMM batch.
- float `predictAMM` (`budgetedData` *testData, `parameters` *param, `budgetedModelAMM` *model, vector< int > *labels=NULL, vector< float > *scores=NULL)
Given a multi-hyperplane machine (MM) model, predict the labels of testing data.

6.5.1 Detailed Description

Defines classes and functions used for training and testing of large-scale multi-hyperplane algorithms (AMM batch, AMM online, and Pegasos).

6.5.2 Function Documentation

6.5.2.1 predictAMM()

```
float predictAMM (
    budgetedData * testData,
    parameters * param,
    budgetedModelAMM * model,
    vector< int > * labels,
    vector< float > * scores )
```

Given a multi-hyperplane machine (MM) model, predict the labels of testing data.

Parameters

in	<i>testData</i>	Input test data.
in	<i>param</i>	The parameters of the algorithm.
in	<i>model</i>	Trained MM model.
out	<i>labels</i>	Vector of predicted labels.
out	<i>scores</i>	Vector of scores of the winning labels.

Returns

Testing set error rate.

Given the learned multi-hyperplane machine model, the function computes the predictions on the testing data, outputting the predicted labels and the error rate.

Definition at line 358 of file mm_algs.cpp.

```

359 {
360     unsigned long N, err = 0, totalPoints = 0;
361     long double fx, maxFx;
362     bool stillChunksLeft = true;
363     long start, timeCalc = 0;
364     char text[1024];
365     budgetedVectorAMM *currentData = NULL;
366     long double *classMaxScores = new long double[(testData->yLabels).size()];
367
368     while (stillChunksLeft)
369     {
370         stillChunksLeft = testData->readChunk((*param).CHUNK_SIZE);
371         (*param).updateVerySparseDataParameter(testData->getSparsity());
372
373         N = testData->N;
374         start = clock();
375         for (unsigned int r = 0; r < N; r++)
376         {
377             totalPoints++;
378             unsigned int y = 0;
379             maxFx = -INF;
380
381             if ((*param).VERY_SPARSE_DATA)
382             {
383                 // compute kernels using vectors directly from the budgetedData
384                 for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
385                 {
386                     classMaxScores[i] = -INF;
387                     for (unsigned int j = 0; j < (*model->getModel())[i].size(); j++)
388                     {
389                         fx = (*(model->getModel())[i][j]->linearKernel(r, testData, param));
390                         if (fx > maxFx)
391                         {
392                             maxFx = fx;
393                             y = i;
394                         }
395
396                         if (fx > classMaxScores[i])
397                             classMaxScores[i] = fx;
398                     }
399                 }
400             }
401             else
402             {
403                 // first create the budgetedVector using the vector from budgetedData, to be used in
404                 gaussianKernel() method below
405                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
406                 currentData->budgetedVector::createVectorUsingDataPoint(testData, r, param);
407
408                 for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
409                 {
410                     classMaxScores[i] = -INF;
411                     for (unsigned int j = 0; j < (*model->getModel())[i].size(); j++)
412                     {
413                         fx = (*(model->getModel())[i][j]->linearKernel(currentData));
414                         if (fx > maxFx)
415                         {
416                             maxFx = fx;
417                             y = i;
418                         }
419
420                         if (fx > classMaxScores[i])
421                             classMaxScores[i] = fx;
422                     }
423                 }
424                 delete currentData;
425                 currentData = NULL;
426             }
427
428             // save predicted label, will be sent to output ...
429             if (labels)
430                 (*labels).push_back((int) (testData->yLabels)[y]);
431             // ... and the scores
432             if (scores)
433             {
434                 // for AMM models the score is the difference between winning and the second best score
435                 long double secondBestScore = -INF;
436                 for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
437                 {
438                     if (i == y)
439                         continue;
440
441                     if (secondBestScore < classMaxScores[i])
442                         secondBestScore = classMaxScores[i];
443                 }
444                 (*scores).push_back((float) (maxFx - secondBestScore));
445             }
446         }
447     }

```



```

445         if (y != testData->al[r])
446             err++;
447     }
448
449     timeCalc += clock() - start;
450
451     if ((*param).VERBOSE) && (N > 0)
452     {
453         sprintf(text, "Number of examples processed: %ld\n", totalPoints);
454         svmPrintString(text);
455     }
456 }
457 delete[] classMaxScores;
458 testData->flushData();
459
460 if ((*param).VERBOSE)
461 {
462     sprintf(text, "*** Testing completed in %5.3f seconds\n*** Testing error rate: %3.2f
percent\n\n", (double)timeCalc / (double)CLOCKS_PER_SEC, 100.0 * (double)err / (double)totalPoints);
463     svmPrintString(text);
464 }
465
466 return (float) (100.0 * (float)err / (float)totalPoints);
467 }

```

6.5.2.2 trainAMMbatch()

```

void trainAMMbatch (
    budgetedData * trainData,
    parameters * param,
    budgetedModelAMM * model )

```

Train AMM batch.

Parameters

in	<i>trainData</i>	Input training data.
in	<i>param</i>	The parameters of the algorithm.
in, out	<i>model</i>	Initial AMM model.

The function trains multi-hyperplane machine using AMM batch algorithm, given input data, the initial model (most often zero-weight model), and the parameters of the model.

Definition at line 960 of file mm_algs.cpp.

```

961 {
962     vector<unsigned int> n; // stores number of weights per class
963     unsigned long timeCalc = 0, start;
964     long double fx1, fx2, maxFx, assocFx;
965     unsigned int i, j, t, N, il, i2, jl, j2, sizeOfLabels = 0, countNew = 0, countDel = 0, numIter = 0,
currAssign = 0, currAssignID, temp;
966     bool stillChunksLeft;
967     char text[1024];
968     budgetedVectorAMM *currentData = NULL;
969
970     //Initialization phase with algorithm AMM_online
971     stillChunksLeft = true;
972     while (stillChunksLeft)
973     {
974         stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);
975
976         // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
977         // (of course, in the case of AMM, speeds up linear kernel computation)
978         (*param).updateVerySparseDataParameter(trainData->getSparsity());
979
980         // compute observed data dimensionality, where we also account for possible bias term, and check
981         if // we need to expand the current model weights if some new data dimensions were found during
loading
982         temp = trainData->getDataDimensionality() + (int) (param->BIAS_TERM != 0.0);

```

```

983     if ((*param).DIMENSION < temp)
984     {
985         /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION, temp);
986         svmPrintString(text);*/
987         (*model).extendDimensionalityOfModel(temp, param);
988
989         // update the dimensionality
990         (*param).DIMENSION = temp;
991     }
992
993     N = trainData->N;
994     if (numIter == 0)
995     {
996         //Initialize
997         sizeOfyLabels = (unsigned int) trainData->yLabels.size();
998         for (i = 0; i < sizeOfyLabels; i++)
999         {
1000             n.push_back(1);
1001
1002             currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_SIZE);
1003             vector <budgetedVectorAMM*> perClassWeights;
1004             perClassWeights.push_back(currentData);
1005             currentData = NULL;
1006             ((*model).getModel()).push_back(perClassWeights);
1007         }
1008     }
1009     else if (sizeOfyLabels != (unsigned int) trainData->yLabels.size())
1010     {
1011         // if in previous chunks some class wasn't observed, could happen with small chunks or
1012         // just add new zero weights for the new classes
1013         for (i = 0; i < (trainData->yLabels.size() - sizeOfyLabels); i++)
1014         {
1015             currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
1016             vector <budgetedVectorAMM*> perClassWeights;
1017             perClassWeights.push_back(currentData);
1018             currentData = NULL;
1019             ((*model).getModel()).push_back(perClassWeights);
1020         }
1021         sizeOfyLabels = (unsigned int) trainData->yLabels.size();
1022     }
1023
1024     // randomize
1025     vector <unsigned int> tv(N, 0);
1026     unsigned int *assigns = new unsigned int[N];
1027     for (i = 0; i < N; i++)
1028     {
1029         tv[i] = i;
1030     }
1031     if ((*param).RANDOMIZE)
1032         random_shuffle(tv.begin(), tv.end());
1033
1034     start = clock();
1035     for (unsigned int trainIter = 0; trainIter < N; trainIter++)
1036     {
1037         numIter++;
1038         t = tv[trainIter];
1039
1040         if (!(*param).VERY_SPARSE_DATA)
1041         {
1042             // only create currentData if the data is non-sparse, otherwise kernels will be
1043             // computed directly from trainData
1044             currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
1045             currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
1046
1047             // calculate i+, j+
1048             i1 = trainData->al[t];
1049             j1 = 0;
1050             maxFx = -INF;
1051             for (j = 0; j < n[i1]; j++)
1052             {
1053                 if ((*param).VERY_SPARSE_DATA)
1054                     fx1 = ((*model).getModel())[i1][j]->linearKernel(t, trainData, param);
1055                 else
1056                     fx1 = ((*model).getModel())[i1][j]->linearKernel(currentData);
1057
1058                 if (fx1 > maxFx)
1059                 {
1060                     j1 = j;
1061                     maxFx = fx1;
1062                 }
1063             }
1064             fx1 = maxFx;
1065             *(assigns + t) = ((*model).getModel())[i1][j1]->getID();
1066
1067             // calculate i-, j-

```

```

1068         i2 = 0;
1069         j2 = 0;
1070         fx2 = 0;
1071         maxFx = -INF;
1072         for (i = 0; i < sizeOfyLabels; i++)
1073         {
1074             if (i == i1)
1075                 continue;
1076
1077             for (j = 0; j < n[i]; j++)
1078             {
1079                 if ((*param).VERY_SPARSE_DATA)
1080                     fx2 = ((*model).getModel()) [i] [j] -> linearKernel(t, trainData, param);
1081                 else
1082                     fx2 = ((*model).getModel()) [i] [j] -> linearKernel(currentData);
1083
1084                 if (fx2 > maxFx)
1085                 {
1086                     maxFx = fx2;
1087                     i2 = i;
1088                     j2 = j;
1089                 }
1090             }
1091         }
1092         fx2 = maxFx;
1093
1094         // downgrade weight each iteration
1095         for (i = 0; i < sizeOfyLabels; i++)
1096             for (j = 0; j < n[i]; j++)
1097                 ((*model).getModel()) [i] [j] -> downgrade(numIter);
1098
1099         if (1.0 + fx2 - fx1 > 0.0)
1100         {
1101             // we made a misprediction, push negative class further away, and positive closer!
1102             if ((*param).VERY_SPARSE_DATA)
1103             {
1104                 // since we did not create currentData earlier, here we create it to perform
1105                 updates
1106                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
1107                 currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
1108             }
1109
1110             // push the other class further away
1111             ((*model).getModel()) [i2] [j2] -> updateUsingVector(currentData, numIter, -1, param);
1112
1113             // update the true class weight
1114             if (fx1 > 0.0)
1115             {
1116                 // here clone the best weight if the cloning probability allows it
1117                 if ((unsigned int) n[i1] < (*param).BUDGET_SIZE)
1118                 {
1119                     if ((*param).CLONE_PROBABILITY > get_random_probability())
1120                     {
1121                         // clone the winning weight
1122                         budgetedVectorAMM *clonedVector = new budgetedVectorAMM((*param).DIMENSION,
1123                         (*param).CHUNK_WEIGHT);
1124                         clonedVector->createVectorUsingVector ((*model).getModel()) [i1] [j1]);
1125
1126                         // add the new cloned weight to the model
1127                         ((*model).getModel()) [i1].push_back(clonedVector);
1128                         n[i1]++;
1129                         clonedVector = NULL;
1130
1131                         // update the clone probability after successful cloning
1132                         (*param).CLONE_PROBABILITY *= (*param).CLONE_PROBABILITY_DECAY;
1133                     }
1134                 }
1135
1136                 ((*model).getModel()) [i1] [j1] -> updateUsingVector(currentData, numIter, 1,
1137                 param);
1138
1139                 delete currentData;
1140                 currentData = NULL;
1141             }
1142             else
1143             {
1144                 if ((unsigned int) n[i1] < (*param).BUDGET_SIZE)
1145                 {
1146                     n[i1]++;
1147                     currentData->updateDegradation(numIter, param);
1148                     ((*model).getModel()) [i1].push_back(currentData);
1149                     currentData = NULL;
1150                     countNew++;
1151                 }
1152                 else
1153                 {
1154                     delete currentData;

```

```

1152         currentData = NULL;
1153     }
1154 }
1155 }
1156 else
1157 {
1158     if (!(*param).VERY_SPARSE_DATA)
1159     {
1160         // if sparse data then no need for this part, since we didn't even create
currentData
1161         delete currentData;
1162         currentData = NULL;
1163     }
1164 }
1165 }
1166 timeCalc += clock() - start;
1167
1168 trainData->saveAssignment(assigns);
1169 delete [] assigns;
1170
1171 if ((*param).VERBOSE) && (N > 0)
1172 {
1173     sprintf(text, "Number of examples processed: %d\n", numIter);
1174     svmPrintString(text);
1175 }
1176 }
1177
1178 if ((*param).VERBOSE)
1179     svmPrintString("Initialization epoch done!\n");
1180
1181 // end of init phase, start AMM algorithm below
1182
1183 for (unsigned int epoch = 1; epoch <= (*param).NUM_EPOCHS; epoch++)
1184 {
1185     stillChunksLeft = true;
1186     while (stillChunksLeft)
1187     {
1188         stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE, true);
1189
1190         // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
1191         // (of course, in the case of AMM, speeds up linear kernel computation)
1192         (*param).updateVerySparseDataParameter(trainData->getSparsity());
1193
1194         // compute observed data dimensionality, where we also account for possible bias term, and
check if
1195         // we need to expand the current model weights if some new data dimensions were found
during loading
1196         temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
1197         if ((*param).DIMENSION < temp)
1198         {
1199             /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION,
temp);
1200             svmPrintString(text);*/
1201             (*model).extendDimensionalityOfModel(temp, param);
1202
1203             // update the dimensionality
1204             (*param).DIMENSION = temp;
1205         }
1206
1207         N = trainData->N;
1208         trainData->readChunkAssignments(!stillChunksLeft);
1209
1210         // randomize
1211         vector<int> tv(N, 0);
1212         for (unsigned int ti = 0; ti < N; ti++)
1213             tv[ti] = ti;
1214
1215         if ((*param).RANDOMIZE)
1216             random_shuffle(tv.begin(), tv.end());
1217
1218         start = clock();
1219         for (unsigned int trainIter = 0; trainIter < N; trainIter++)
1220         {
1221             numIter++;
1222             t = tv[trainIter];
1223
1224             if (!(*param).VERY_SPARSE_DATA)
1225             {
1226                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
1227                 currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
1228             }
1229
1230             // calculate i+, j+
1231             i1 = trainData->a1[t];
1232             j1 = 0;
1233
1234             currAssignID = trainData->assignments[t];

```

```

1235
1236         maxFx = 0;
1237         assocFx = -INF;
1238         for (j = 0; j < n[i1]; j++)
1239         {
1240             if ((*param).VERY_SPARSE_DATA)
1241                 fx1 = ((*model).getModel())[i1][j]->linearKernel(t, trainData, param);
1242             else
1243                 fx1 = ((*model).getModel())[i1][j]->linearKernel(currentData);
1244
1245             if ((maxFx == 0) || (fx1 > maxFx))
1246             {
1247                 j1 = j;
1248                 maxFx = fx1;
1249             }
1250
1251             // this is the prediction of the associated same-label weight
1252             if ((*model).getModel())[i1][j]->getID() == currAssignID
1253             {
1254                 currAssign = j;
1255                 assocFx = fx1;
1256             }
1257         }
1258
1259         fx1 = maxFx;
1260         if (assocFx == -INF)
1261         {
1262             assocFx = maxFx;
1263             currAssign = j1;
1264         }
1265
1266         // calculate i-, j-
1267         i2 = 0;
1268         j2 = 0;
1269         fx2 = 0;
1270         maxFx = 0;
1271         for (i = 0; i < sizeOfyLabels; i++)
1272         {
1273             if (i == i1)
1274                 continue;
1275
1276             for (j = 0; j < n[i]; j++)
1277             {
1278                 if ((*param).VERY_SPARSE_DATA)
1279                     fx2 = ((*model).getModel())[i][j]->linearKernel(t, trainData, param);
1280                 else
1281                     fx2 = ((*model).getModel())[i][j]->linearKernel(currentData);
1282
1283                 if ((maxFx == 0) || (fx2 > maxFx))
1284                 {
1285                     maxFx = fx2;
1286                     i2 = i;
1287                     j2 = j;
1288                 }
1289             }
1290         }
1291         fx2 = maxFx;
1292
1293         // downgrade weights each iteration
1294         for (unsigned int i = 0; i < sizeOfyLabels; i++)
1295         {
1296             for (unsigned int j = 0; j < n[i]; j++)
1297             {
1298                 ((*model).getModel())[i][j]->downgrade(numIter);
1299             }
1300         }
1301
1302         // calculate v
1303         if (1.0 + fx2 - assocFx > 0.0)
1304         {
1305             // we made a misprediction, update the weights by pushing the wrong-class weight
1306             further from the misclassified
1307             // example, and the true-class closer to the misclassified example
1308             if ((*param).VERY_SPARSE_DATA)
1309             {
1310                 // since we did not create currentData earlier, here we create it to perform
1311                 updates
1312                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
1313                 currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
1314             }
1315
1316             // duplicate the assigned true-class weight if the cloning probability allows it
1317             if ((assocFx > 0.0) && ((unsigned int)n[i1] < (*param).BUDGET_SIZE))
1318             {
1319                 if ((*param).CLONE_PROBABILITY > get_random_probability())
1320                 {
1321                     // clone the associated weight

```

```

1320         budgetedVectorAMM *clonedVector = new budgetedVectorAMM((*param).DIMENSION,
1321         (*param).CHUNK_WEIGHT);
1322         clonedVector->createVectorUsingVector ((*((*model).getModel()))[i1][currAssign]);
1323         // add the new cloned weight to the model
1324         ((*((*model).getModel()))[i1].push_back(clonedVector);
1325         n[i1]++;
1326         clonedVector = NULL;
1327         // update the clone probability after successful cloning
1328         (*param).CLONE_PROBABILITY *= (*param).CLONE_PROBABILITY_DECAY;
1329     }
1330 }
1331 }
1332 }
1333     ((*((*model).getModel()))[i1][currAssign]->updateUsingVector(currentData, numIter,
1334     l, param);
1335     ((*((*model).getModel()))[i2][j2]->updateUsingVector(currentData, numIter, -1,
1336     param);
1337     if ((fx1 <= 0.0) && (n[i1] < (*param).BUDGET_SIZE))
1338     {
1339         n[i1]++;
1340         currentData->updateDegradation(numIter, param);
1341         ((*((*model).getModel()))[i1].push_back(currentData);
1342         currentData = NULL;
1343         countNew++;
1344     }
1345     else
1346     {
1347         // if over the budget, we do not add a new data point to the budget
1348         delete currentData;
1349         currentData = NULL;
1350     }
1351 }
1352 else
1353 {
1354     if (!(*param).VERY_SPARSE_DATA)
1355     {
1356         // if sparse data then no need for this part, since we didn't even create
1357         currentData
1358         delete currentData;
1359         currentData = NULL;
1360     }
1361 }
1362 timeCalc += clock() - start;
1363 start = clock();
1364 if (numIter % (*param).K_PARAM == 0)
1365 {
1366     // we run the pruning procedure here
1367     long double sumNorms = 0;
1368     long double sumThreshold = (long double)(*param).C_PARAM * (long
1369     double)(*param).C_PARAM / ((long double)numIter * (long double)numIter * (*param).LAMBDA_PARAM *
1370     (*param).LAMBDA_PARAM);
1371     vector<long double> weightNorms, sortedWeightNorms;
1372     int numToDelete = 0;
1373     // first find the norms of weights
1374     for (unsigned int i = 0; i < sizeOfyLabels; i++)
1375     {
1376         for (vector<budgetedVectorAMM*>::iterator vi =
1377         ((*((*model).getModel()))[i].begin(); vi != ((*((*model).getModel()))[i].end(); vi++)
1378         {
1379             weightNorms.push_back((double) ((*vi)).getSqrL2norm());
1380         }
1381     }
1382     // now sort them
1383     sortedWeightNorms = weightNorms;
1384     sort(sortedWeightNorms.begin(), sortedWeightNorms.end());
1385     // find how many before threshold is exceeded
1386     for (unsigned int i = 0; i < weightNorms.size(); i++)
1387     {
1388         sumNorms += sortedWeightNorms[i];
1389         if (sumNorms > sumThreshold)
1390             break;
1391         else
1392             numToDelete++;
1393     }
1394     // delete those that should be deleted, with aggregate norm less than the set
1395     threshold
1396     int counter = 0;
1397     bool deleted = false;
1398     for (unsigned int i = 0; i < sizeOfyLabels; i++)

```

```

1398         {
1399             for (vector<budgetedVectorAMM*>::iterator vi =
1400                 ((*model).getModel())[i].begin(); vi != ((*model).getModel())[i].end();)
1401             {
1402                 long double currNorm = weightNorms[counter++];
1403                 deleted = false;
1404                 for (int j = 0; j < numToDelete; j++)
1405                 {
1406                     if (currNorm == sortedWeightNorms[j])
1407                     {
1408                         if (n[i] == 1)
1409                         {
1410                             svmPrintString("Was about to delete all weights of a class,
1411 check the K_PARAM and C_PARAM parameters!\n");
1412                             break;
1413                         }
1414                         delete (*vi);
1415                         vi = ((*model).getModel())[i].erase(vi);
1416                         n[i]--;
1417                         countDel++;
1418                         deleted = true;
1419                         break;
1420                     }
1421                 }
1422             }
1423             if (!deleted)
1424                 vi++;
1425         }
1426     }
1427 }
1428 }
1429 }
1430 timeCalc += clock() - start;
1431 if (((*param).VERBOSE) && (N > 0))
1432 {
1433     sprintf(text, "Number of examples processed: %d\n", numIter);
1434     svmPrintString(text);
1435 }
1436 }
1437 }
1438 }
1439 // every so-so (around 3) subepochs recalculate the associations
1440 if ((epoch % (*param).NUM_SUBEPOCHS) == 0) && (epoch != (*param).NUM_EPOCHS))
1441 {
1442     // calculate the new assignments
1443     stillChunksLeft = true;
1444     while (stillChunksLeft)
1445     {
1446         stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);
1447         N = trainData->N;
1448         unsigned int *assigns = new unsigned int[N];
1449         start = clock();
1450         for (unsigned int ot = 0; ot < N; ot++)
1451         {
1452             t = ot;
1453             if ((*param).VERY_SPARSE_DATA)
1454             {
1455                 // calculate i+, j+
1456                 i1 = trainData->al[t];
1457                 j1 = 0;
1458                 maxFx = -INF;
1459                 for (unsigned int j = 0; j < n[i1]; j++)
1460                 {
1461                     fx1 = ((*model).getModel())[i1][j]->linearKernel(t, trainData, param);
1462                     if (fx1 > maxFx)
1463                     {
1464                         j1 = j;
1465                         maxFx = fx1;
1466                     }
1467                 }
1468             }
1469             else
1470             {
1471                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
1472                 currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
1473                 // calculate i+, j+
1474                 i1 = trainData->al[t];
1475                 j1 = 0;
1476                 maxFx = -INF;
1477                 for (unsigned int j = 0; j < n[i1]; j++)
1478                 {
1479                     fx1 = ((*model).getModel())[i1][j]->linearKernel(currentData);
1480                     if (fx1 > maxFx)
1481                     {
1482                         j1 = j;
1483                     }
1484                 }
1485             }
1486             assigns[ot] = j1;
1487         }
1488     }
1489 }

```

```

1483         {
1484             j1 = j;
1485             maxFx = fx1;
1486         }
1487     }
1488     delete currentData;
1489     currentData = NULL;
1490 }
1491
1492     * (assigns + t) = ((*model).getModel()) [i1][j1]->getID();
1493 }
1494     timeCalc += clock() - start;
1495
1496     trainData->saveAssignment(assigns);
1497     delete [] assigns;
1498 }
1499 }
1500
1501     if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
1502     {
1503         sprintf(text, "Epoch %d/%d done.\n", epoch, (*param).NUM_EPOCHS);
1504         svmPrintString(text);
1505     }
1506 }
1507 trainData->flushData();
1508
1509     if ((*param).VERBOSE)
1510     {
1511         sprintf(text, "*** Training completed in %5.3f seconds.\nNumber of weights deleted: %d\n",
(double) timeCalc / (double) CLOCKS_PER_SEC, countDel);
1512         svmPrintString(text);
1513         for (unsigned int i = 0; i < sizeOfyLabels; i++)
1514         {
1515             sprintf(text, "Number of weights of class %d: %d\n", i + 1, n[i]);
1516             svmPrintString(text);
1517         }
1518     }
1519 }

```

6.5.2.3 trainAMMonline()

```

void trainAMMonline (
    budgetedData * trainData,
    parameters * param,
    budgetedModelAMM * model )

```

Train AMM online.

Parameters

in	<i>trainData</i>	Input training data.
in	<i>param</i>	The parameters of the algorithm.
in, out	<i>model</i>	Initial AMM model.

The function trains multi-hyperplane machine using AMM online algorithm, given input data, the initial model (most often zero-weight model), and the parameters of the model.

Definition at line 651 of file mm_algs.cpp.

```

652 {
653     vector<unsigned int> n;
654     unsigned long timeCalc = 0, start;
655     long double fx1, fx2, maxFx;
656     unsigned int sizeOfyLabels = 0, countNew = 0, countDel = 0, numIter = 0, i1, i2, j1, j2, t, N, temp;
657     bool stillChunksLeft = true;
658     char text[1024];
659     budgetedVectorAMM *currentData = NULL;
660
661     // train the model
662     for (unsigned int epoch = 0; epoch < (*param).NUM_EPOCHS; epoch++)

```



```

663     {
664         stillChunksLeft = true;
665         while (stillChunksLeft)
666         {
667             stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);
668
669             // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
670             // (of course, in the case of AMM, speeds up linear kernel computation)
671             (*param).updateVerySparseDataParameter(trainData->getSparsity());
672
673             // compute observed data dimensionality, where we also account for possible bias term, and
674             // we need to expand the current model weights if some new data dimensions were found
675             // during loading
676             temp = trainData->getDataDimensionality() + (int) (param->BIAS_TERM != 0.0);
677             if ((*param).DIMENSION < temp)
678             {
679                 /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION,
680                 temp);
681                 svmPrintString(text);*/
682                 (*model).extendDimensionalityOfModel(temp, param);
683
684                 // update the dimensionality
685                 (*param).DIMENSION = temp;
686             }
687
688             N = trainData->N;
689             if (numIter == 0)
690             {
691                 // initialize the model with zero weights
692                 sizeOfLabels = (unsigned int) trainData->yLabels.size();
693                 for (unsigned int i = 0; i < sizeOfLabels; i++)
694                 {
695                     n.push_back(1);
696
697                     currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
698                     vector <budgetedVectorAMM*> v1;
699                     v1.push_back(currentData);
700                     currentData = NULL;
701                     ((*model).getModel()).push_back(v1);
702                 }
703             }
704             else if (sizeOfLabels != trainData->yLabels.size())
705             {
706                 // if in the chunks before some class wasn't observed, could happen with small chunks or
707                 // unbalanced classes
708                 for (unsigned int i = 0; i < (trainData->yLabels.size() - sizeOfLabels); i++)
709                 {
710                     currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
711                     vector <budgetedVectorAMM*> perClassWeights;
712                     perClassWeights.push_back(currentData);
713                     currentData = NULL;
714                     ((*model).getModel()).push_back(perClassWeights);
715                 }
716                 sizeOfLabels = (unsigned int) trainData->yLabels.size();
717             }
718
719             // randomize
720             vector <unsigned int> tv(N, 0);
721             for (unsigned int ti = 0; ti < N; ti++)
722             {
723                 tv[ti] = ti;
724             }
725             if ((*param).RANDOMIZE)
726                 random_shuffle(tv.begin(), tv.end());
727
728             start = clock();
729             for (unsigned int ot = 0; ot < N; ot++)
730             {
731                 numIter++;
732                 t = tv[ot];
733
734                 if (!(*param).VERY_SPARSE_DATA)
735                 {
736                     // only create currentData if the data is non-sparse, otherwise kernels will be
737                     // computed directly from trainData
738                     currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
739                     currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
740                 }
741
742                 //calculate i+,j+
743                 i1 = trainData->al[t];
744                 j1 = 0;
745                 maxFx = -INF;
746                 for (unsigned int j = 0; j < n[i1]; j++)
747                 {

```

```

745         if ((*param).VERY_SPARSE_DATA)
746             fx1 = ((*model).getModel()) [i1][j]->linearKernel(t, trainData, param);
747         else
748             fx1 = ((*model).getModel()) [i1][j]->linearKernel(currentData);
749
750         if (fx1 > maxFx)
751         {
752             j1 = j;
753             maxFx = fx1;
754         }
755     }
756     fx1 = maxFx;
757
758     // calculate i-, j-
759     i2 = 0;
760     j2 = 0;
761     fx2 = 0;
762     maxFx = -INF;
763     for (unsigned int i = 0; i < sizeOfyLabels; i++)
764     {
765         if (i == i1)
766             continue;
767
768         for (unsigned int j = 0; j < n[i]; j++)
769         {
770             if ((*param).VERY_SPARSE_DATA)
771                 fx2 = ((*model).getModel()) [i][j]->linearKernel(t, trainData, param);
772             else
773                 fx2 = ((*model).getModel()) [i][j]->linearKernel(currentData);
774
775             if (fx2 > maxFx)
776             {
777                 maxFx = fx2;
778                 i2 = i;
779                 j2 = j;
780             }
781         }
782     }
783     fx2 = maxFx;
784
785     // downgrade weights each iteration
786     for (unsigned int i = 0; i < sizeOfyLabels; i++)
787         for (unsigned int j = 0; j < n[i]; j++)
788             ((*model).getModel()) [i][j]->downgrade(numIter);
789
790     if (1.0 + fx2 - fx1 > 0.0)
791     {
792         // we made a misprediction, push negative class further away, and positive closer!
793
794         if ((*param).VERY_SPARSE_DATA)
795         {
796             // since we did not create currentData earlier, here we create it to perform
797             updates
798             currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
799             currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
800
801             // push the other class further away
802             param);
803             ((*model).getModel()) [i2][j2]->updateUsingVector(currentData, numIter, -1,
804
805             // update the true class weight
806             if (fx1 > 0.0)
807             {
808                 // here clone the best weight if the cloning probability allows it
809                 if ((unsigned int)n[i1] < (*param).BUDGET_SIZE)
810                 {
811                     if ((*param).CLONE_PROBABILITY > get_random_probability())
812                     {
813                         // clone the winning weight
814                         budgetedVectorAMM *clonedVector = new
815                         budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
816                         clonedVector->createVectorUsingVector ((*model).getModel()) [i1][j1]);
817
818                         // add the new cloned weight to the model
819                         ((*model).getModel()) [i1].push_back(clonedVector);
820                         n[i1]++;
821                         clonedVector = NULL;
822
823                         // update the clone probability after successful cloning
824                         (*param).CLONE_PROBABILITY *= (*param).CLONE_PROBABILITY_DECAY;
825                     }
826                 }
827             }
828             ((*model).getModel()) [i1][j1]->updateUsingVector(currentData, numIter, 1,
829
830             param);

```

```

827         delete currentData;
828         currentData = NULL;
829     }
830     else
831     {
832         if (n[i1] < (*param).BUDGET_SIZE) // limit number of weights (we found ~20 is a
reasonable number per class)
833         {
834             n[i1]++;
835             currentData->updateDegradation(numIter, param);
836             ((*model).getModel())[i1].push_back(currentData);

837             countNew++;
838             currentData = NULL;
839         }
840         else
841         {
842             delete currentData;
843             currentData = NULL;
844         }
845     }
846 }
847 else
848 {
849     if (!(*param).VERY_SPARSE_DATA)
850     {
851         // if sparse data then no need for this part, since we didn't even create
currentData
852         delete currentData;
853         currentData = NULL;
854     }
855 }
856
857 // pruning phase
858 if (numIter % (int) (*param).K_PARAM == 0)
859 {
860     long double sumNorms = 0, sumThreshold = (long double) (*param).C_PARAM * (long
double) (*param).C_PARAM / ((long double) numIter * (long double) numIter * (*param).LAMBDA_PARAM *
(*param).LAMBDA_PARAM);
861     vector<long double> weightNorms, sortedWeightNorms;
862     int numToDelete = 0;
863
864     // first find the norms of weights
865     for (unsigned int i = 0; i < sizeOfLabels; i++)
866     {
867         for (vector<budgetedVectorAMM*>::iterator vi =
(*((*model).getModel())[i].begin(); vi != ((*model).getModel())[i].end(); vi++)
868         {
869             weightNorms.push_back((double) ((*vi)).getSqrL2norm());
870         }
871     }
872
873     // now sort them
874     sortedWeightNorms = weightNorms;
875     sort(sortedWeightNorms.begin(), sortedWeightNorms.end());
876
877     // find how many before threshold exceeded
878     for (unsigned int i = 0; i < weightNorms.size(); i++)
879     {
880         sumNorms += sortedWeightNorms[i];
881         if (sumNorms > sumThreshold)
882             break;
883         else
884             numToDelete++;
885     }
886
887     // delete those that should be deleted
888     int counter = 0;
889     bool deleted = false;
890     for (unsigned int i = 0; i < sizeOfLabels; i++)
891     {
892         for (vector<budgetedVectorAMM*>::iterator vi =
(*((*model).getModel())[i].begin(); vi != ((*model).getModel())[i].end(); vi++)
893         {
894             long double currNorm = weightNorms[counter++];
895
896             deleted = false;
897             for (int j = 0; j < numToDelete; j++)
898             {
899                 if (currNorm == sortedWeightNorms[j])
900                 {
901                     if (n[i] == 1)
902                     {
903                         svmPrintString("Was about to delete all weights of a class,
check K_PARAM and C_PARAM parameters!\n");
904                         break;
905                     }

```

```

906
907         delete (*vi);
908         vi = ((*model).getModel())[i].erase(vi);
909         n[i]--;
910
911         countDel++;
912         deleted = true;
913         break;
914     }
915 }
916
917     if (!deleted)
918         vi++;
919 }
920 }
921 }
922 }
923 timeCalc += clock() - start;
924
925 if ((*param).VERBOSE) && (N > 0)
926 {
927     sprintf(text, "Number of examples processed: %d\n", numIter);
928     svmPrintString(text);
929 }
930 }
931
932 if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
933 {
934     sprintf(text, "Epoch %d/%d done.\n", epoch + 1, (*param).NUM_EPOCHS);
935     svmPrintString(text);
936 }
937 }
938 trainData->flushData();
939
940 if ((*param).VERBOSE)
941 {
942     sprintf(text, "*** Training completed in %5.3f seconds.\nNumber of weights deleted: %d\n",
943 (double)timeCalc / (double)CLOCKS_PER_SEC, countDel);
944     svmPrintString(text);
945     for (unsigned int i = 0; i < sizeOfyLabels; i++)
946     {
947         sprintf(text, "Number of weights of class %d: %d\n", i + 1, n[i]);
948         svmPrintString(text);
949     }
950 }

```

6.5.2.4 trainPegasos()

```

void trainPegasos (
    budgetedData * trainData,
    parameters * param,
    budgetedModelAMM * model )

```

Train Pegasos.

Parameters

in	<i>trainData</i>	Input training data.
in	<i>param</i>	The parameters of the algorithm.
in, out	<i>model</i>	Initial Pegasos model.

The function trains Pegasos model, given input data, initial model (most often zero-weight model), and the parameters of the model.

Definition at line 477 of file mm_algs.cpp.

```

478 {
479     unsigned int sizeOfyLabels = 0, numIter = 0, t, i1, i2 = 0, N, temp;

```

```

480     unsigned long timeCalc = 0, start;
481     long double fx, fx1, fx2, maxFx;
482     bool stillChunksLeft = true;
483     char text[1024];
484     budgetedVectorAMM *currentData = NULL;
485
486     // train the model
487     for (unsigned int epoch = 0; epoch < (*param).NUM_EPOCHS; epoch++)
488     {
489         stillChunksLeft = true;
490         while (stillChunksLeft)
491         {
492             stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);
493
494             // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
495             // (of course, in the case of AMM, speeds up linear kernel computation)
496             (*param).updateVerySparseDataParameter(trainData->getSparsity());
497
498             // compute observed data dimensionality, where we also account for possible bias term, and
499             // we need to expand the current model weights if some new data dimensions were found
500             temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
501             if ((*param).DIMENSION < temp)
502             {
503                 /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION,
504                 temp);
505                 svmPrintString(text);*/
506                 (*model).extendDimensionalityOfModel(temp, param);
507
508                 // update the dimensionality
509                 (*param).DIMENSION = temp;
510             }
511             N = trainData->N;
512             if (numIter == 0)
513             {
514                 // initialize the model
515                 sizeOfyLabels = (unsigned int) trainData->yLabels.size();
516                 for (unsigned int i = 0; i < sizeOfyLabels; i++)
517                 {
518                     currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
519                     vector <budgetedVectorAMM*> perClassWeights;
520                     perClassWeights.push_back(currentData);
521                     currentData = NULL;
522                     ((*model).getModel()).push_back(perClassWeights);
523                 }
524             }
525             else if (sizeOfyLabels != (unsigned int) trainData->yLabels.size())
526             {
527                 // if in the chunks before some class wasn't observed add it here; could happen with
528                 // small chunks or unbalanced classes
529                 for (unsigned int i = 0; i < (trainData->yLabels.size() - sizeOfyLabels); i++)
530                 {
531                     currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
532                     vector <budgetedVectorAMM*> perClassWeights;
533                     perClassWeights.push_back(currentData);
534                     currentData = NULL;
535                     ((*model).getModel()).push_back(perClassWeights);
536                 }
537                 sizeOfyLabels = (unsigned int) trainData->yLabels.size();
538             }
539             vector <unsigned int> tv(N, 0);
540             for (unsigned int ti = 0; ti < N; ti++)
541             {
542                 tv[ti] = ti;
543             }
544
545             // randomize the data
546             if ((*param).RANDOMIZE)
547                 random_shuffle(tv.begin(), tv.end());
548
549             start = clock();
550             for (unsigned int ot = 0; ot < N; ot++)
551             {
552                 numIter++;
553                 t = tv[ot];
554
555                 il = trainData->al[t];
556                 if ((*param).VERY_SPARSE_DATA)
557                 {
558                     // compute kernels using vectors directly from the budgetedData
559                     fx1 = ((*model).getModel())[il][0]->linearKernel(t, trainData, param);
560                 }
561                 else
562                 {

```

```

563         // first create the budgetedVector using the vector from budgetedData, to be used in
linearKernel() method below
564         currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
565         currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
566
567         fx1 = ((*model).getModel())[i1][0]->linearKernel(currentData);
568     }
569
570     //calculate i-, fi-
571     fx = 0;
572     maxFx = -INF;
573     for (unsigned int i = 0; i < sizeOfyLabels; i++)
574     {
575         if (i == i1)
576             continue;
577
578         if ((*param).VERY_SPARSE_DATA)
579             fx = ((*model).getModel())[i][0]->linearKernel(t, trainData, param);
580         else
581             fx = ((*model).getModel())[i][0]->linearKernel(currentData);
582
583         if (fx > maxFx)
584         {
585             maxFx = fx;
586             i2 = i;
587         }
588     }
589     fx2 = maxFx;
590
591     // downgrade the weights
592     for (unsigned int i = 0; i < sizeOfyLabels; i++)
593     {
594         ((*model).getModel())[i][0]->downgrade(numIter);
595     }
596
597     // calculate the margin, if misclassified update weights
598     if (1.0L + fx2 - fx1 > 0.0L)
599     {
600         if ((*param).VERY_SPARSE_DATA)
601         {
602             ((*model).getModel())[i2][0]->updateUsingDataPoint(trainData, numIter, t, -1,
param);
603             ((*model).getModel())[i1][0]->updateUsingDataPoint(trainData, numIter, t, 1,
param);
604         }
605         else
606         {
607             ((*model).getModel())[i2][0]->updateUsingVector(currentData, numIter, -1,
param);
608             ((*model).getModel())[i1][0]->updateUsingVector(currentData, numIter, 1,
param);
609         }
610     }
611
612     if (!(*param).VERY_SPARSE_DATA)
613     {
614         // if sparse data then no need for this part, since we didn't even create
currentData
615         delete currentData;
616         currentData = NULL;
617     }
618     timeCalc += clock() - start;
619
620     if ((*param).VERBOSE) && (N > 0)
621     {
622         sprintf(text, "Number of examples processed: %d\n", numIter);
623         svmPrintString(text);
624     }
625
626     if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
627     {
628         sprintf(text, "Epoch %d/%d done.\n", epoch + 1, (*param).NUM_EPOCHS);
629         svmPrintString(text);
630     }
631     trainData->flushData();
632
633     if ((*param).VERBOSE)
634     {
635         sprintf(text, "*** Training completed in %5.3f seconds.\n", (double)timeCalc /
(double)CLOCKS_PER_SEC);
636         svmPrintString(text);
637     }
638 }
639 }
640 }
641 }

```

Index

- alphaNorm
 - budgetedVectorBSGD, 94
- alphas
 - budgetedVectorBSGD, 96
- array
 - budgetedVector, 82
- arrayLength
 - budgetedVector, 82
- assignments
 - budgetedData, 20
- BIAS_TERM
 - parameters, 100
- bsgd.h
 - predictBSGD, 118
 - trainBSGD, 120
- BUDGET_SIZE
 - parameters, 101
- budgetedData, 9
 - assignments, 20
 - budgetedData, 11, 12
 - dimensionHighestSeen, 20
 - distanceBetweenTwoPoints, 13
 - fAssignFile, 20
 - getDataDimensionality, 14
 - getElementOfVector, 14
 - getNumLoadedDataPointsSoFar, 15
 - getSparsity, 15
 - getVectorSqrL2Norm, 16
 - ifileNameAssign, 21
 - readChunk, 17
 - readChunkAssignments, 19
 - saveAssignment, 19
- budgetedDataMatlab, 21
 - budgetedDataMatlab, 22
 - readChunk, 23
 - readDataFromMatlab, 23
- budgetedModel, 26
 - extendDimensionalityOfModel, 27
 - getAlgorithm, 27
 - loadFromTextFile, 28
 - saveToTextFile, 29
- budgetedModelAMM, 30
 - extendDimensionalityOfModel, 31
 - getModel, 31
 - loadFromTextFile, 31
 - saveToTextFile, 33
- budgetedModelBSGD, 35
 - extendDimensionalityOfModel, 36
 - loadFromTextFile, 36
 - saveToTextFile, 38
- budgetedModelLLSVM, 40
 - extendDimensionalityOfModel, 41
 - loadFromTextFile, 41
 - saveToTextFile, 44
- budgetedModelMatlab, 45
 - getAlgorithm, 46
 - loadFromMatlabStruct, 46
 - saveToMatlabStruct, 47
- budgetedModelMatlabAMM, 48
 - loadFromMatlabStruct, 49
 - saveToMatlabStruct, 51
- budgetedModelMatlabBSGD, 54
 - loadFromMatlabStruct, 54
 - saveToMatlabStruct, 56
- budgetedModelMatlabLLSVM, 58
 - loadFromMatlabStruct, 59
 - saveToMatlabStruct, 61
- budgetedSVM.h
 - fgetWord, 126
 - parseInputPrompt, 127
 - printUsagePrompt, 133
 - readableFileExists, 135
 - setPrintErrorStringFunction, 136
 - setPrintStringFunction, 136
 - svmPrintErrorString, 137
 - svmPrintString, 137
- budgetedSVM_matlab.h
 - fakeAnswer, 108
 - parseInputMatlab, 108
 - printErrorStringMatlab, 114
 - printStringMatlab, 115
 - printUsageMatlab, 115
- budgetedVector, 63
 - array, 82
 - arrayLength, 82
 - budgetedVector, 66
 - chunkWeight, 82
 - computeKernel, 66, 67
 - createVectorUsingDataPoint, 68
 - createVectorUsingVector, 69
 - exponentialKernel, 69, 70
 - extendDimensionality, 71
 - gaussianKernel, 72, 73
 - getDimensionality, 73
 - getID, 74
 - getSqrL2norm, 74
 - id, 83
 - linearKernel, 74, 75

- operator[], [76](#), [77](#)
- polyKernel, [77](#), [78](#)
- setSqrL2norm, [78](#)
- sigmoidKernel, [79](#)
- sqrL2norm, [83](#)
- sqrNorm, [80](#)
- userDefinedKernel, [80](#), [81](#)
- weightID, [83](#)
- budgetedVectorAMM, [84](#)
 - budgetedVectorAMM, [85](#)
 - createVectorUsingDataPoint, [86](#)
 - createVectorUsingVector, [86](#)
 - degradation, [91](#)
 - downgrade, [87](#)
 - getDegradation, [87](#)
 - getSqrL2norm, [87](#)
 - linearKernel, [88](#)
 - sqrNorm, [89](#)
 - updateDegradation, [89](#)
 - updateUsingDataPoint, [90](#)
 - updateUsingVector, [90](#)
- budgetedVectorBSGD, [92](#)
 - alphaNorm, [94](#)
 - alphas, [96](#)
 - budgetedVectorBSGD, [93](#)
 - downgrade, [94](#)
 - getNumClasses, [95](#)
 - updateSV, [95](#)
- budgetedVectorLLSVM, [97](#)
 - createVectorUsingDataPointMatrix, [97](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h, [107](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h, [117](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h, [124](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h, [138](#)
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h, [142](#)
- C_PARAM
 - parameters, [101](#)
- CHUNK_SIZE
 - parameters, [101](#)
- CHUNK_WEIGHT
 - parameters, [102](#)
- chunkWeight
 - budgetedVector, [82](#)
- CLONE_PROBABILITY
 - parameters, [102](#)
- computeKernel
 - budgetedVector, [66](#), [67](#)
- createVectorUsingDataPoint
 - budgetedVector, [68](#)
 - budgetedVectorAMM, [86](#)
- createVectorUsingDataPointMatrix
 - budgetedVectorLLSVM, [97](#)
- createVectorUsingVector
 - budgetedVector, [69](#)
 - budgetedVectorAMM, [86](#)
- degradation
 - budgetedVectorAMM, [91](#)
- DIMENSION
 - parameters, [102](#)
- dimensionHighestSeen
 - budgetedData, [20](#)
- distanceBetweenTwoPoints
 - budgetedData, [13](#)
- downgrade
 - budgetedVectorAMM, [87](#)
 - budgetedVectorBSGD, [94](#)
- exponentialKernel
 - budgetedVector, [69](#), [70](#)
- extendDimensionality
 - budgetedVector, [71](#)
- extendDimensionalityOfModel
 - budgetedModel, [27](#)
 - budgetedModelAMM, [31](#)
 - budgetedModelBSGD, [36](#)
 - budgetedModelLLSVM, [41](#)
- fakeAnswer
 - budgetedSVM_matlab.h, [108](#)
- fAssignFile
 - budgetedData, [20](#)
- fgetWord
 - budgetedSVM.h, [126](#)
- gaussianKernel
 - budgetedVector, [72](#), [73](#)
- getAlgorithm
 - budgetedModelMatlab, [46](#)
- getDimensionality
 - budgetedData, [14](#)
- getDegradation
 - budgetedVectorAMM, [87](#)
- getDimensionality
 - budgetedVector, [73](#)
- getElementOfVector
 - budgetedData, [14](#)
- getID
 - budgetedVector, [74](#)
- getModel
 - budgetedModelAMM, [31](#)
- getNumClasses
 - budgetedVectorBSGD, [95](#)
- getNumLoadedDataPointsSoFar
 - budgetedData, [15](#)
- getSparsity
 - budgetedData, [15](#)
- getSqrL2norm
 - budgetedVector, [74](#)
 - budgetedVectorAMM, [87](#)
- getVectorSqrL2Norm

- budgetedData, 16
- id
 - budgetedVector, 83
- ifileNameAssign
 - budgetedData, 21
- K_MEANS_ITERS
 - parameters, 103
- K_PARAM
 - parameters, 103
- KERNEL
 - parameters, 103
- LAMBDA_PARAM
 - parameters, 104
- linearKernel
 - budgetedVector, 74, 75
 - budgetedVectorAMM, 88
- llsvm.h
 - predictLLSVM, 138
 - trainLLSVM, 140
- loadFromMatlabStruct
 - budgetedModelMatlab, 46
 - budgetedModelMatlabAMM, 49
 - budgetedModelMatlabBSGD, 54
 - budgetedModelMatlabLLSVM, 59
- loadFromTextFile
 - budgetedModel, 28
 - budgetedModelAMM, 31
 - budgetedModelBSGD, 36
 - budgetedModelLLSVM, 41
- MAINTENANCE_SAMPLING_STRATEGY
 - parameters, 104
- mm_algs.h
 - predictAMM, 143
 - trainAMMbatch, 145
 - trainAMMonline, 152
 - trainPegasos, 156
- NUM_EPOCHS
 - parameters, 105
- NUM_SUBEPOCHS
 - parameters, 105
- operator[]
 - budgetedVector, 76, 77
- OUTPUT_SCORES
 - parameters, 105
- parameters, 98
 - BIAS_TERM, 100
 - BUDGET_SIZE, 101
 - C_PARAM, 101
 - CHUNK_SIZE, 101
 - CHUNK_WEIGHT, 102
 - CLONE_PROBABILITY, 102
 - DIMENSION, 102
 - K_MEANS_ITERS, 103
 - K_PARAM, 103
 - KERNEL, 103
 - LAMBDA_PARAM, 104
 - MAINTENANCE_SAMPLING_STRATEGY, 104
 - NUM_EPOCHS, 105
 - NUM_SUBEPOCHS, 105
 - OUTPUT_SCORES, 105
 - updateVerySparseDataParameter, 100
 - VERY_SPARSE_DATA, 106
- parseInputMatlab
 - budgetedSVM_matlab.h, 108
- parseInputPrompt
 - budgetedSVM.h, 127
- polyKernel
 - budgetedVector, 77, 78
- predictAMM
 - mm_algs.h, 143
- predictBSGD
 - bsgd.h, 118
- predictLLSVM
 - llsvm.h, 138
- printErrorStringMatlab
 - budgetedSVM_matlab.h, 114
- printStringMatlab
 - budgetedSVM_matlab.h, 115
- printUsageMatlab
 - budgetedSVM_matlab.h, 115
- printUsagePrompt
 - budgetedSVM.h, 133
- readableFileExists
 - budgetedSVM.h, 135
- readChunk
 - budgetedData, 17
 - budgetedDataMatlab, 23
- readChunkAssignments
 - budgetedData, 19
- readDataFromMatlab
 - budgetedDataMatlab, 23
- saveAssignment
 - budgetedData, 19
- saveToMatlabStruct
 - budgetedModelMatlab, 47
 - budgetedModelMatlabAMM, 51
 - budgetedModelMatlabBSGD, 56
 - budgetedModelMatlabLLSVM, 61
- saveToTextFile
 - budgetedModel, 29
 - budgetedModelAMM, 33
 - budgetedModelBSGD, 38
 - budgetedModelLLSVM, 44
- setPrintErrorStringFunction
 - budgetedSVM.h, 136
- setPrintStringFunction
 - budgetedSVM.h, 136
- setSqrL2norm
 - budgetedVector, 78
- sigmoidKernel

- [budgetedVector](#), [79](#)
- [sqrL2norm](#)
 - [budgetedVector](#), [83](#)
- [sqrNorm](#)
 - [budgetedVector](#), [80](#)
 - [budgetedVectorAMM](#), [89](#)
- [svmPrintErrorString](#)
 - [budgetedSVM.h](#), [137](#)
- [svmPrintString](#)
 - [budgetedSVM.h](#), [137](#)
- [trainAMMbatch](#)
 - [mm_algs.h](#), [145](#)
- [trainAMMonline](#)
 - [mm_algs.h](#), [152](#)
- [trainBSGD](#)
 - [bsgd.h](#), [120](#)
- [trainLLSVM](#)
 - [llsvm.h](#), [140](#)
- [trainPegasos](#)
 - [mm_algs.h](#), [156](#)
- [updateDegradation](#)
 - [budgetedVectorAMM](#), [89](#)
- [updateSV](#)
 - [budgetedVectorBSGD](#), [95](#)
- [updateUsingDataPoint](#)
 - [budgetedVectorAMM](#), [90](#)
- [updateUsingVector](#)
 - [budgetedVectorAMM](#), [90](#)
- [updateVerySparseDataParameter](#)
 - [parameters](#), [100](#)
- [userDefinedKernel](#)
 - [budgetedVector](#), [80](#), [81](#)
- [VERY_SPARSE_DATA](#)
 - [parameters](#), [106](#)
- [weightID](#)
 - [budgetedVector](#), [83](#)