BudgetedSVM

1.2

Generated by Doxygen 1.8.18

Sun Jun 28 2020 02:22:51

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:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

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

Chapter 5

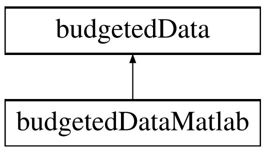
Class Documentation

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	*keepAssignments*	True for AMM batch, otherwise false. File 'temp_assigns.txt' will be created and deleted to keep the assignments.
in	*yLabels*	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->al = 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	*fileName*	Path to the input .txt file.
in	*dimension*	Dimensionality of the classification problem.
in	*chunkSize*	Size of the input data chunk that is loaded.
in	*keepAssignments*	True for AMM batch, otherwise false. File 'temp_assigns.txt' will be created and deleted to keep the assignments.
in	*yLabels*	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
    dimensionality during data loading
246         this->dimensionHighestSeen = 0;
247     else
248         this->dimensionHighestSeen = dimension;
249
250     this->al = new (nothrow) unsigned char[chunkSize];
251     if (this->al == NULL)
252     {
253         svmPrintErrorString("Memory allocation error (budgetedData Constructor)!");
254     }
255
256     // keepAssignments is used for AMM_batch, where we hold the epoch assignments of data points to
    hyperplanes
257     this->keepAssignments = keepAssignments;
258     if (keepAssignments)
```

```
    {
        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
        this->assignments = new (nothrow) unsigned int[chunkSize];
    }
    else
        this->assignments = NULL;

    // if labels provided use them, this happens in the case of testing data
    if (yLabels)
    {
        for (unsigned int i = 0; i < (*yLabels).size(); i++)
        {
            this->yLabels.push_back((*yLabels)[i]);
        }
        this->isTrainingSet = false;
    }

    this->fileOpened = false;
    this->fileAssignOpened = false;
    this->loadTime = 0;
    this->N = 0;
    this->dataPartiallyLoaded = true;
    this->loadedDataPointsSoFar = 0;
    this->numNonZeroFeatures = 0;
}
```

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	*index1*	Index of the first data point.
in	*index2*	Index of the second data point.

Returns

Euclidean distance between the two points.

Definition at line 580 of file budgetedSVM.cpp.

```
{
    // if distance to itself, return 0.0
    if (index1 == index2)
        return 0.0;

    long icurrent1 = ai[index1];
    long iend1 = (index1 == ai.size() - 1) ? aj.size() : ai[index1 + 1];
    long icurrent2 = ai[index2];
    long iend2 = (index2 == ai.size() - 1) ? aj.size() : ai[index2 + 1];
    double dotxx = 0.0, dotyy = 0.0, dotxy = 0.0;

    double currFeat1, currFeat2;
    while (1)
    {
        // traverse the vectors non-zero feature by non-zero feature
        if (icurrent1 < iend1)
            currFeat1 = (double) aj[icurrent1];
```

```
        else
            currFeat1 = INF;
        if (icurrent2 < iend2)
            currFeat2 = (double) aj[icurrent2];
        else
            currFeat2 = INF;

        if (currFeat1 == currFeat2)
        {
            dotxy += (an[icurrent1] * an[icurrent2]);
            dotxx += (an[icurrent1] * an[icurrent1]);
            dotyy += (an[icurrent2] * an[icurrent2]);

            icurrent1++;
            icurrent2++;
        }
        else
        {
            if (currFeat1 < currFeat2)
            {
                dotxx += (an[icurrent1] * an[icurrent1]);
                icurrent1++;
            }
            else
            {
                dotyy += (an[icurrent2] * an[icurrent2]);
                icurrent2++;
            }
        }

        if ((icurrent1 >= iend1) && (icurrent2 >= iend2))
            break;
    }
    return dotxx + dotyy - 2.0 * dotxy;
}
```

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.

```
        {
            return dimensionHighestSeen;
        };
```

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

```
{
    unsigned int maxPointIndex, pointIndexPointer;

    // check if vector index too big
    if (vector >= this->N)
    {
        svmPrintString("Warning: Vector index in getElementOfVector() function out of bounds, returning default value of 0.\n");
        return 0.0;
    }
    // check if element index too big
    if (element >= this->dimensionHighestSeen)
    {
        svmPrintString("Warning: Element index in getElementOfVector() function out of bounds, returning default value of 0.\n");
        return 0.0;
    }

    pointIndexPointer = this->ai[vector];
    maxPointIndex = ((unsigned int)(vector + 1) == this->N) ? (unsigned int) (this->aj.size()) : this->ai[vector + 1];

    for (unsigned int i = pointIndexPointer; i < maxPointIndex; i++)
    {
        // if we found the element return its value
        if (this->aj[i] == element + 1)
            return this->an[i];

        // if we went over the index of the wanted element, then the element is equal to 0
        if (this->aj[i] > element + 1)
            return 0.0;
    }
    // if the wanted element is indexed higher than all non-zero elements, then it is equal to 0
    return 0.0;
}
```

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.

```
        {
            return loadedDataPointsSoFar;
        };
```

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.

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

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	*vector*	Index of the vector (C-style indexing used, starting from 0; note that LibSVM format indices start from 1).
in	*param*	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.

```
{
    unsigned int maxPointIndex, pointIndexPointer;
    long double result = 0.0;

    // check if vector index too big
    if (vector >= this->N)
    {
        svmPrintString("Warning: Vector index in getElementOfVector() function out of bounds, returning
      default value of 0.\n");
        return 0.0;
    }

    pointIndexPointer = this->ai[vector];
    maxPointIndex = ((unsigned int)(vector + 1) == this->N) ? (unsigned int)(this->aj.size()) :
      this->ai[vector + 1];

    for (unsigned int i = pointIndexPointer; i < maxPointIndex; i++)
```

```
            result += (this->an[i] * this->an[i]);
    if (param->BIAS_TERM != 0.0)
        result += (param->BIAS_TERM * param->BIAS_TERM);

    return result;
}
```

5.1.3.7 readChunk()

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

Reads the next data chunk.

Parameters

in	*size*	Size of the chunk (i.e., number of data points) to be loaded.
in	*assign*	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.

```
{
    string text;

    char line[262143];  // maximum length of the line to be read is set to 262143
    char str[256];
    int pos, label;
    unsigned int counter = 0, dimSeen, pointIndex = 0;
    unsigned long start = clock();
    bool labelFound, warningWritten = false;

    // if not loaded from .txt file just exit
    if (!dataPartiallyLoaded)
        return false;

    flushData();
    if (!fileOpened)
    {
        this->ifile = fopen(ifileName, "rt");
        this->fileOpened = true;
        this->loadedDataPointsSoFar = 0;
        this->numNonZeroFeatures = 0;

        // if the very beginning, just create the assignment file if necessary
        if ((!assign) && (keepAssignments))
        {
            fAssignFile = fopen(ifileNameAssign, "wt");
            fclose(fAssignFile);
        }
    }

```

```
    // load chunk
    while (fgets(line, 262143, ifile))
    {
        N++;
        loadedDataPointsSoFar++;

        stringstream ss;
        ss « line;

        // get label
        if (ss » text)
        {
            label = atoi(text.c_str());
            ai.push_back(pointIndex);

            // get yLabels, if label not seen before add it into the label array
            labelFound = false;
            for (unsigned int i = 0; i < yLabels.size(); i++)
            {
                if (yLabels[i] == label)
                {
                    al[counter++] = (char) i;
                    labelFound = true;
                    break;
                }
            }

            if (!labelFound)
            {
                if (isTrainingSet)
                {
                    yLabels.push_back(label);
                    al[counter++] = (char) (yLabels.size() - 1);
                }
                else
                {
                    // so unseen label detected during testing phase, issue a warning
                    if (!warningWritten)
                    {
                        sprintf(str, "Warning: Testing label '%d' detected during loading that was not seen in training.\n", label);
                        svmPrintString(str);
                        warningWritten = true;
                    }

                    // give an example a label index that can never be predicted
                    al[counter++] = (char) yLabels.size();
                }
            }
        }

        // get feature values
        while (ss » text)
        {
            if ((pos = (int) text.find(":")))
            {
                dimSeen = atoi(text.substr(0, pos).c_str());
                aj.push_back(dimSeen);
                an.push_back((float) atof(text.substr(pos + 1, text.length()).c_str()));
                pointIndex++;
                numNonZeroFeatures++;

                // if more features found than specified, print error message
                /*if (dimensionHighestSeen < dimSeen)
                {
                    sprintf(line, "Found more features than specified with '-D' option (specified: %d, found %d)!\nPlease check your settings.\n", dimension, dimSeen);
                    svmPrintErrorString(line);
                }*/

                if (dimensionHighestSeen < dimSeen)
                    dimensionHighestSeen = dimSeen;
            }
        }

        // check the size of chunk
        if (N == size)
        {
            // still data left to load, keep working
            loadTime += (clock() - start);
            return true;
        }
    }

    // got to the end of file, no more data left to load, exit nicely
    fclose(ifile);
    fileOpened = false;
```

```
    loadTime += (clock() - start);

    return false;
}
```

5.1.3.8 readChunkAssignments()

```
void budgetedData::readChunkAssignments (
            bool endOfFile )
```

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

Parameters

in	*endOfFile*	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 to expensive to store all assignments in memory when working with large data sets.

Definition at line 334 of file budgetedSVM.cpp.

```
{
    // if data is fully loaded from the beginning then just exit (e.g., can happen when BudgetedSVM is called from Matlab interface)
    if (!dataPartiallyLoaded)
        return;

    int tempInt;
    if (!fileAssignOpened)
    {
        fileAssignOpened = true;
        fAssignFile = fopen(ifileNameAssign, "rt");
    }

    for (unsigned int i = 0; i < N; i++)
    {
        // get the assignments (as opposed to initial iteration and reassignment phase
        // where we write the assignments, here we read them)
        if (!fscanf(fAssignFile, "%d\n", &tempInt))
        {
            svmPrintErrorString("Error reading assignments from the text file!\n");
        }
        *(assignments + i) = (unsigned int) tempInt;
    }

    if (endOfFile)
    {
        fileAssignOpened = false;
        fclose(fAssignFile);
    }
};
```

5.1.3.9 saveAssignment()

```
void budgetedData::saveAssignment (
            unsigned int * assigns )
```

Saves the current assignments, used by AMM batch.

Parameters

in	*assigns*	Current assignments.

Definition at line 308 of file budgetedSVM.cpp.

```
{
    // 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)
    if (!dataPartiallyLoaded)
    {
        if (assignments == NULL)
            assignments = new (nothrow) unsigned int[N];

        for (unsigned int i = 0; i < N; i++)
            *(assignments + i) = *(assigns + i);

        return;
    }

    fAssignFile = fopen(ifileNameAssign, "at");

    for (unsigned int i = 0; i < N; i++)
        fprintf(fAssignFile, "%d\n", *(assigns + i));

    fclose(fAssignFile);
};
```

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:

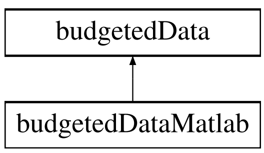
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.cpp

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	*labelVec*	Vector of labels.
in	*instanceMat*	Matrix of data points, each row is a single data point.
in	*param*	The parameters of the algorithm.
in	*keepAssignments*	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	*yLabels*	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	*size*	Size of the chunk to be loaded.
in	*assign*	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	*labelVec*	Vector of labels.
in	*instanceMat*	Matrix of data points, each row is a single data point.
in	*param*	The parameters of the algorithm.

Definition at line 82 of file budgetedSVM_matlab.cpp.

```
{
    long start = clock();
    unsigned int i, j, k, labelVectorRowNum;
    long unsigned int low, high;
    mwIndex *ir, *jc;
    double *samples, *labels;
    bool labelFound;
    mxArray *instanceMatCol; // transposed instance sparse matrix
    bool warningWritten = false;
    char str[256];

    // otherwise load the data, given below
    // transpose instance matrix
    {
        mxArray *prhs[1], *plhs[1];
        prhs[0] = mxDuplicateArray(instanceMat);
        if (mexCallMATLAB(1, plhs, 1, prhs, "transpose"))
             mexErrMsgTxt("Error: Cannot transpose training instance matrix.\n");

         instanceMatCol = plhs[0];
         mxDestroyArray(prhs[0]);
     }

     // each column is one instance
     labels = mxGetPr(labelVec);
     samples = mxGetPr(instanceMatCol);

     // get number of instances
     labelVectorRowNum = (int)mxGetM(labelVec);
     if (labelVectorRowNum != (int)mxGetN(instanceMatCol))
         mexErrMsgTxt("Length of label vector does not match number of instances.\n");

     // set the dimension and the number of data points
     this->N = labelVectorRowNum;
     if ((*param).DIMENSION == 0)
     {
         // it is 0 when loading training data set
         this->dimensionHighestSeen = (*param).DIMENSION = (int)mxGetM(instanceMatCol);
         if ((*param).BIAS_TERM != 0.0)
             (*param).DIMENSION++;

         // set KERNEL_GAMMA_PARAM here if needed, done during loading of training set
         if ((*param).KERNEL_GAMMA_PARAM == 0.0)
             (*param).KERNEL_GAMMA_PARAM = 1.0 / (double) (*param).DIMENSION;
     }
     else
     {
         // it is non-zero only when loading testing data set, no need to set GAMMA parameter as it is read from the model structure from Matlab
         this->dimensionHighestSeen = (*param).DIMENSION;

         // if bias term is non-zero, then the actual dimensionality of data is one less than DIMENSION
         if ((*param).BIAS_TERM != 0.0)
             this->dimensionHighestSeen--;
     }

     // allocate memory for labels
     this->al = new (nothrow) unsigned char[this->N];
     if (this->al == NULL)
         mexErrMsgTxt("Memory allocation error (readDataFromMatlab function)! Restart MATLAB and try again.");

     if (mxIsSparse(instanceMat))
     {
         ir = mxGetIr(instanceMatCol);
         jc = mxGetJc(instanceMatCol);

         j = 0;
         for (i = 0; i < labelVectorRowNum; i++)
         {
             // where the instance starts
             ai.push_back(j);

             // get yLabels, if label not seen before add it in the label array
             labelFound = false;
             for (k = 0; k < (int) yLabels.size(); k++)
             {
                 if (yLabels[k] == (int)labels[i])
                 {
                     al[i] = k;
                     labelFound = true;
                     break;
                 }
             }
             if (!labelFound)
             {
```

```
                if (isTrainingSet)
                {
                    yLabels.push_back((int)labels[i]);
                    al[i] = (unsigned char) (yLabels.size() - 1);
                }
                else
                {
                    // so unseen label detected during testing phase, issue a warning
                    if (!warningWritten)
                    {
                        sprintf(str, "Warning: Testing label '%d' detected that was not seen during training.\n", (int)labels[i]);
                        mexPrintf(str);
                        mexEvalString("drawnow;");

                        warningWritten = true;
                    }

                    // give an example a label index that can never be predicted
                    al[i] = (unsigned char) yLabels.size();
                }
            }

            // get features
            low = (int) jc[i], high = (int) jc[i + 1];
            for (k = low; k < high; k++)
            {
                // we save the actual feature no. in aj, and the value in an
                aj.push_back((int) ir[k] + 1);
                an.push_back((float) samples[k]);
                j++;
            }
        }
    }
    else
    {
        j = 0;
        low = 0;
        for (i = 0; i < labelVectorRowNum; i++)
        {
            // where the instance starts
            ai.push_back(j);

            // get yLabels, if label not seen before add it in the label array
            labelFound = false;
            for (k = 0; k < (int) yLabels.size(); k++)
            {
                if (yLabels[k] == (int) labels[i])
                {
                    al[i] = k;
                    labelFound = true;
                    break;
                }
            }
            if (!labelFound)
            {
                if (isTrainingSet)
                {
                    yLabels.push_back((int)labels[i]);
                    al[i] = (unsigned char) (yLabels.size() - 1);
                }
                else
                {
                    // so unseen label detected during testing phase, issue a warning
                    if (!warningWritten)
                    {
                        sprintf(str, "Warning: Testing label '%d' detected that was not seen during training.\n", (int) labels[i]);
                        mexPrintf(str);
                        mexEvalString("drawnow;");

                        warningWritten = true;
                    }

                    // give an example a label index that can never be predicted
                    al[i] = (unsigned char) yLabels.size();
                }
            }

            // get features
            for (k = 0; k < (int)mxGetM(instanceMatCol); k++)
            {
                if (samples[low] != 0.0)
                {
                    // we save the actual feature no. in aj, and the value in an
                    aj.push_back(k + 1);
                    an.push_back((float) samples[low]);
```

```
                    j++;
                }
                low++;
            }
        }
    }

    // if very beginning, just allocate memory for assignments
    if (keepAssignments)
        this->assignments = new (nothrow) unsigned int[this->N];

    loadTime += (clock() - start);
};
```

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

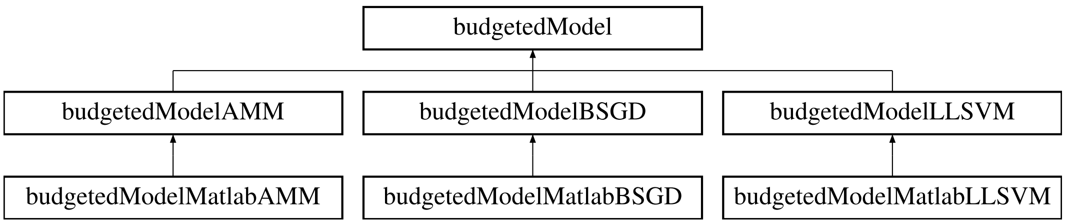
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.cpp

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	*newDim*	Filename of the .txt file where the model is saved.
in	*param*	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	*filename*	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.

```
{
    FILE *fModel = NULL;
    int temp;
    fModel = fopen(filename, "rt");
    if (!fModel)
        return -1;

    if (!fscanf(fModel, "ALGORITHM: %d\n", &temp))
    {
        svmPrintErrorString("Error reading algorithm type from the model file!\n");
    }
    return temp;
}
```

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	*filename*	Filename of the .txt file where the model is saved.
out	*yLabels*	Vector of possible labels.
out	*param*	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	*filename*	Filename of the .txt file where the model is saved.
in	*yLabels*	Vector of possible labels.
in	*param*	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:

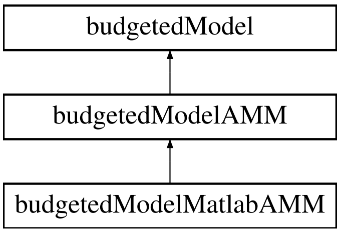
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.cpp

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.

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

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.

```
        {
            return modelMM;
        };
```

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	*filename*	Filename of the .txt file where the model is saved.
out	*yLabels*	Vector of possible labels.
out	*param*	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.

```
{
    unsigned int i, j, tempInt, numClasses;
    float tempFloat;
    string text;
    char oneWord[1024];
    int pos;
    vector <unsigned int> numWeights;
    FILE *fModel = NULL;
    fModel = fopen(filename, "rt");
    bool doneReadingBool;
    long double sqrNorm;

    if (!fModel)
        return false;

    // algorithm
    fseek (fModel, (long) strlen("ALGORITHM: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &((*param).ALGORITHM)))
    {
        svmPrintErrorString("Error reading algorithm type from the model file!\n");
    }

    // dimension
    fseek (fModel, (long) strlen("DIMENSION: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &((*param).DIMENSION)))
    {
        svmPrintErrorString("Error reading dimensions from the model file!\n");
    }

    // number of classes
    fseek (fModel, (long) strlen("NUMBER_OF_CLASSES: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &numClasses))
    {
        svmPrintErrorString("Error reading number of classes from the model file!\n");
    }

    // labels
    fseek (fModel, (long) strlen("LABELS: "), SEEK_CUR);
    for (i = 0; i < numClasses; i++)
    {
        if (!fscanf(fModel, "%d ", &tempInt))
        {
            svmPrintErrorString("Error reading labels from the model file!\n");
        }
        (*yLabels).push_back(tempInt);
    }

    // number of weights
    fseek (fModel, (long) strlen("NUMBER_OF_WEIGHTS: "), SEEK_CUR);
    for (i = 0; i < numClasses; i++)
    {
        if (!fscanf(fModel, "%d\n", &tempInt))
        {
```

```
            svmPrintErrorString("Error reading number of weights from the model file!\n");
        }
        numWeights.push_back(tempInt);
    }

    // bias parameter
    fseek(fModel, (long) strlen("BIAS_TERM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading bias term from the model file!\n");
    }
    (*param).BIAS_TERM = tempFloat;

    // kernel width (GAMMA) parameter
    fseek (fModel, (long) strlen("KERNEL_WIDTH: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading kernel width from the model file!\n");
    }
    (*param).KERNEL_GAMMA_PARAM = tempFloat;

    // load the model
    fseek (fModel, (long) strlen("MODEL:\n") + 1, SEEK_CUR);
    for (i = 0; i < numClasses; i++)                                // for every class
    {
        // add for each class an empty weight matrix
        vector <budgetedVectorAMM*> tempV;
        (*modelMM).push_back(tempV);

        for (j = 0; j < numWeights[i]; j++)                         // for every weight
        {
            budgetedVectorAMM *eNew = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
            sqrNorm = 0.0L;

            // get degradation and features

            // skip label, no need to read it explicitly since we know the number of weights of each
      class, found in numWeights vector
            fgetWord(fModel, oneWord);

            // get degradation
            doneReadingBool = fgetWord(fModel, oneWord);
            eNew->setDegradation((long double) atof(oneWord));

            // get features
            while (!doneReadingBool)
            {
                doneReadingBool = fgetWord(fModel, oneWord);
                if (strlen(oneWord) == 0)
                    continue;

                text = oneWord;
                if ((pos = (int) text.find(":")))
                {
                    tempInt = atoi(text.substr(0, pos).c_str());
                    tempFloat = (float) atof(text.substr(pos + 1, text.length()).c_str());
                    (*eNew)[tempInt - 1] = tempFloat;

                    sqrNorm += (long double)(tempFloat * tempFloat);
                }
            }
            eNew->setSqrL2norm(sqrNorm);

            (*modelMM)[i].push_back(eNew);
            eNew = NULL;
        }
    }

    fclose(fModel);
    return true;
}
```

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	*filename*	Filename of the .txt file where the model is saved.
in	*yLabels*	Vector of possible labels.
in	*param*	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.

```
{
    unsigned int i, j, k;
    FILE *fModel = NULL;
    fModel = fopen(filename, "wt");

    if (!fModel)
        return false;

    // algorithm
    fprintf(fModel, "ALGORITHM: %d\n", (*param).ALGORITHM);

    // dimension
    fprintf(fModel, "DIMENSION: %d\n", (*param).DIMENSION);

    // number of classes
    fprintf(fModel, "NUMBER_OF_CLASSES: %d\n", (int) (*yLabels).size());

    // labels
    fprintf(fModel, "LABELS:");
    for (i = 0; i < (*yLabels).size(); i++)
        fprintf(fModel, " %d", (*yLabels)[i]);
    fprintf(fModel, "\n");

    // number of weights
    fprintf(fModel, "NUMBER_OF_WEIGHTS:");
    for (i = 0; i < (*modelMM).size(); i++)
        fprintf(fModel, " %d", (int) (*modelMM)[i].size());
    fprintf(fModel, "\n");

    // bias parameter
    fprintf(fModel, "BIAS_TERM: %f\n", (*param).BIAS_TERM);

    // kernel width (GAMMA) parameter
    fprintf(fModel, "KERNEL_WIDTH: 0.0\n");

    // save the model
    fprintf(fModel, "MODEL:\n");
    for (i = 0; i < yLabels->size(); i++)                // for every class
    {
        for (j = 0; j < (*modelMM)[i].size(); j++)       // for every weight
        {
            // weight label
            fprintf(fModel, "%d ", (*yLabels)[i]);

            // degradation
            fprintf(fModel, "%2.10f", (double)((*modelMM)[i][j])->getDegradation());

            for (k = 0; k < (*param).DIMENSION; k++)     // for every feature
            {
                if ((*((*modelMM)[i][j]))[k] != 0.0)
                    fprintf(fModel, " %d:%2.10f", k + 1, (*((*modelMM)[i][j]))[k]);
            }
            fprintf(fModel, "\n");
```

```
        }
    }

    fclose(fModel);
    return true;
}
```

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

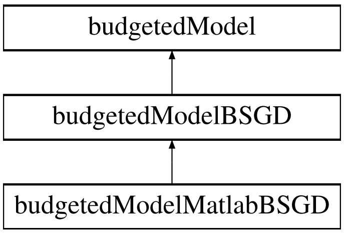
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.cpp

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.

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

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	*filename*	Filename of the .txt file where the model is saved.
out	*yLabels*	Vector of possible labels.
out	*param*	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.

```
{
    unsigned int i, numClasses;
    float tempFloat;
    string text;
    char oneWord[1024];
    int pos, tempInt;
    vector <unsigned int> numWeights;
    FILE *fModel = NULL;
    fModel = fopen(filename, "rt");
    bool doneReadingBool;
    long double sqrNorm;

    if (!fModel)
        return false;

    // algorithm
    fseek (fModel, strlen("ALGORITHM: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &((*param).ALGORITHM)))
    {
        svmPrintErrorString("Error reading algorithm type from the model file!\n");
    }

    // dimension
    fseek (fModel, strlen("DIMENSION: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &((*param).DIMENSION)))
    {
        svmPrintErrorString("Error reading dimensions from the model file!\n");
    }

    // number of classes
    fseek (fModel, strlen("NUMBER_OF_CLASSES: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &numClasses))
    {
        svmPrintErrorString("Error reading number of classes from the model file!\n");
    }

    // labels
    fseek (fModel, strlen("LABELS: "), SEEK_CUR);
    for (i = 0; i < numClasses; i++)
    {
        if (!fscanf(fModel, "%d ", &tempInt))
        {
            svmPrintErrorString("Error reading labels from the model file!\n");
        }
        (*yLabels).push_back(tempInt);
    }

    // number of weights
    fseek (fModel, strlen("NUMBER_OF_WEIGHTS: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &tempInt))
    {
        svmPrintErrorString("Error reading number of weight from the model file!\n");
    }
    numWeights.push_back(tempInt);

    // bias parameter
    fseek (fModel, strlen("BIAS_TERM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading bias term from the model file!\n");
    }
    (*param).BIAS_TERM = tempFloat;
```

```

    // read kernel function
    fseek (fModel, strlen("KERNEL_FUNCTION: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &tempInt))
    {
        svmPrintErrorString("Error reading kernel function type from the model file!\n");
    }
    (*param).KERNEL = tempInt;

    // kernel parameter (GAMMA) parameter
    fseek (fModel, strlen("KERNEL_GAMMA_PARAM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading RBF kernel width parameter from the model file!\n");
    }
    (*param).KERNEL_GAMMA_PARAM = tempFloat;

    // kernel degree/slope parameter
    fseek (fModel, strlen("KERNEL_DEGREE_PARAM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading kernel degree/slope parameter from the model file!\n");
    }
    (*param).KERNEL_DEGREE_PARAM = tempFloat;

    // kernel coefficient/intercept parameter
    fseek (fModel, strlen("KERNEL_COEF_PARAM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading kernel coefficient/intercept parameter from the model
file!\n");
    }
    (*param).KERNEL_COEF_PARAM = tempFloat;

    // load the model
    fseek (fModel, strlen("MODEL:\n") + 1, SEEK_CUR);

    for (i = 0; i < numWeights[0]; i++)                              // for every weight
    {
        budgetedVectorBSGD *eNew = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT,
numClasses);
        sqrNorm = 0.0L;

        // get alphas and features
        doneReadingBool = false;
        while (!doneReadingBool)
        {
            doneReadingBool = fgetWord(fModel, oneWord);
            if (strlen(oneWord) == 0)
                continue;

            text = oneWord;
            if ((pos = (int) text.find(":")))
            {
                tempInt = atoi(text.substr(0, pos).c_str());
                tempFloat = (float) atof(text.substr(pos + 1, text.length()).c_str());

                // alphas have negative index, features have positive
                if (tempInt > 0)
                {
                    (*eNew)[tempInt - 1] = tempFloat;
                    sqrNorm += (long double)(tempFloat * tempFloat);
                }
                else
                    (*eNew).alphas[- tempInt - 1] = tempFloat;
            }
        }
        eNew->setSqrL2norm(sqrNorm);

        (*modelBSGD).push_back(eNew);
        eNew = NULL;
    }

    fclose(fModel);
    return true;
}
```

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	*filename*	Filename of the .txt file where the model is saved.
in	*yLabels*	Vector of possible labels.
in	*param*	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.

```
{
    unsigned int i, j;
    FILE *fModel = NULL;
    fModel = fopen(filename, "wt");
    bool tempBool;

    if (!fModel)
        return false;

    // algorithm
    fprintf(fModel, "ALGORITHM: %d\n", (*param).ALGORITHM);

    // dimension
    fprintf(fModel, "DIMENSION: %d\n", (*param).DIMENSION);

    // number of classes
    fprintf(fModel, "NUMBER_OF_CLASSES: %d\n", (int) (*yLabels).size());

    // labels
    fprintf(fModel, "LABELS:");
    for (i = 0; i < (*yLabels).size(); i++)
        fprintf(fModel, " %d", (*yLabels)[i]);
    fprintf(fModel, "\n");

    // number of weights
    fprintf(fModel, "NUMBER_OF_WEIGHTS: ");
    fprintf(fModel, "%d\n", (int) (*modelBSGD).size());

    // bias parameter
    fprintf(fModel, "BIAS_TERM: %f\n", (*param).BIAS_TERM);

    // kernel function
    fprintf(fModel, "KERNEL_FUNCTION: %d\n", (*param).KERNEL);

    // kernel width (GAMMA) parameter
    fprintf(fModel, "KERNEL_GAMMA_PARAM: %f\n", (*param).KERNEL_GAMMA_PARAM);

    // kernel degree/slope parameter
    fprintf(fModel, "KERNEL_DEGREE_PARAM: %f\n", (*param).KERNEL_DEGREE_PARAM);

     // kernel coef/intercept parameter
     fprintf(fModel, "KERNEL_COEF_PARAM: %f\n", (*param).KERNEL_COEF_PARAM);

```

```
    // save the model
    fprintf(fModel, "MODEL:\n");
    for (i = 0; i < (*modelBSGD).size(); i++)
    //for (i = 0; i < 50; i++)
    {
        for (j = 0; j < (*yLabels).size(); j++)
        {
            // alphas have negative index to differentiate them from features
            if ((*((*modelBSGD)[i])).alphas[j] != 0.0)
                fprintf(fModel, "-%d:%2.10f ", j + 1, (double)(*((*modelBSGD)[i])).alphas[j]);
        }

        // this tempBool is used so that the line doesn't end with a white-space, it makes our life
        // easier when reading word-by-word from the model file using fgetWord(); we can, of course,
        // do without it, but to avoid unnecessary checks during loading of the model we do it here
        tempBool = true;
        for (j = 0; j < (*param).DIMENSION; j++)                    // for every feature
        {
            if ((*((*modelBSGD)[i]))[j] != 0.0)
            {
                if (tempBool)
                {
                    fprintf(fModel, "%d:%2.10f", j + 1, (*((*modelBSGD)[i]))[j]);
                    tempBool = false;
                }
                else
                    fprintf(fModel, " %d:%2.10f", j + 1, (*((*modelBSGD)[i]))[j]);
            }
        }
        fprintf(fModel, "\n");
    }

    fclose(fModel);
    return true;
}
```

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

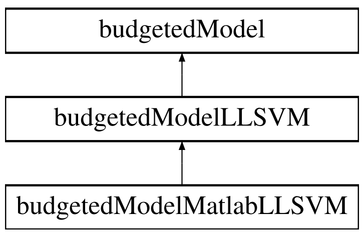
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.cpp

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	*filename*	Filename of the .txt file where the model is saved.
out	*yLabels*	Vector of possible labels.
out	*param*	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     {
```

```
        if (!fscanf(fModel, "%d ", &tempInt))
        {
            svmPrintErrorString("Error reading labels from the model file!\n");
        }
        (*yLabels).push_back(tempInt);
    }

    // number of weights
    fseek (fModel, strlen("NUMBER_OF_WEIGHTS: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &tempInt))
    {
        svmPrintErrorString("Error reading number of weights from the model file!\n");
    }
    (*param).BUDGET_SIZE = tempInt;

    // bias parameter
    fseek (fModel, strlen("BIAS_TERM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading bias term from the model file!\n");
    }
    (*param).BIAS_TERM = tempFloat;

    // read kernel function
    fseek (fModel, strlen("KERNEL_FUNCTION: "), SEEK_CUR);
    if (!fscanf(fModel, "%d\n", &tempInt))
    {
        svmPrintErrorString("Error reading kernel function type from the model file!\n");
    }
    (*param).KERNEL = tempInt;

    // kernel parameter (GAMMA) parameter
    fseek (fModel, strlen("KERNEL_GAMMA_PARAM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading RBF kernel width parameter from the model file!\n");
    }
    (*param).KERNEL_GAMMA_PARAM = tempFloat;

    // kernel degree/slope parameter
    fseek (fModel, strlen("KERNEL_DEGREE_PARAM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading kernel degree/slope parameter from the model file!\n");
    }
    (*param).KERNEL_DEGREE_PARAM = tempFloat;

    // kernel coefficient/intercept parameter
    fseek (fModel, strlen("KERNEL_COEF_PARAM: "), SEEK_CUR);
    if (!fscanf(fModel, "%f\n", &tempFloat))
    {
        svmPrintErrorString("Error reading kernel coefficient/intercept parameter from the model
   file!\n");
    }
    (*param).KERNEL_COEF_PARAM = tempFloat;

    // allocate memory for model
    modelLLSVMmatrixW.resize((*param).BUDGET_SIZE, (*param).BUDGET_SIZE);

    // initialize weight (i.e., hyperplane) in the projected space to zero-vector
    modelLLSVMweightVector.resize((*param).BUDGET_SIZE);

    // load the model
    fseek (fModel, strlen("MODEL:\n") + 1, SEEK_CUR);
    for (i = 0; i < (*param).BUDGET_SIZE; i++)
    {
        budgetedVectorLLSVM *eNew = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
        sqrNorm = 0.0L;

        // get alphas and features below

        // get linear SVM feature
        doneReadingBool = fgetWord(fModel, oneWord);
        modelLLSVMweightVector(i) = (double) atof(oneWord);

        // get elements of modelLLSVMmatrixW
        for (unsigned int j = 0; j < (*param).BUDGET_SIZE; j++)
        {
            doneReadingBool = fgetWord(fModel, oneWord);
            modelLLSVMmatrixW(i, j) = (double) atof(oneWord);
        }

        // get features
        while (!doneReadingBool)
        {
            doneReadingBool = fgetWord(fModel, oneWord);
            if (strlen(oneWord) == 0)
```

```
                continue;

            text = oneWord;
            if ((pos = (int) text.find(":")))
            {
                tempInt = atoi(text.substr(0, pos).c_str());
                tempFloat = (float) atof(text.substr(pos + 1, text.length()).c_str());
                (*eNew)[tempInt - 1] = tempFloat;
                sqrNorm += (long double)(tempFloat * tempFloat);
            }
        }
        eNew->setSqrL2norm(sqrNorm);
        (*modelLLSVMlandmarks).push_back(eNew);
        eNew = NULL;
    }
    fclose(fModel);
    return true;
}
```

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	*filename*	Filename of the .txt file where the model is saved.
in	*yLabels*	Vector of possible labels.
in	*param*	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.

```
{
    unsigned int i, j;
    FILE *fModel = NULL;

    fModel = fopen(filename, "wt");
    if (!fModel)
        return false;

    // algorithm
    fprintf(fModel, "ALGORITHM: %d\n", (*param).ALGORITHM);

    // dimension
    fprintf(fModel, "DIMENSION: %d\n", (*param).DIMENSION);
```

```

    // number of classes
    fprintf(fModel, "NUMBER_OF_CLASSES: %d\n", (int) (*yLabels).size());

    // labels
    fprintf(fModel, "LABELS:");
    for (i = 0; i < (*yLabels).size(); i++)
        fprintf(fModel, " %d", (*yLabels)[i]);
    fprintf(fModel, "\n");

    // number of weights
    fprintf(fModel, "NUMBER_OF_WEIGHTS:");
    fprintf(fModel, " %d\n", (int) (*modelLLSVMlandmarks).size());

    // bias parameter
    fprintf(fModel, "BIAS_TERM: %f\n", (*param).BIAS_TERM);

    // kernel function
    fprintf(fModel, "KERNEL_FUNCTION: %d\n", (*param).KERNEL);

    // kernel width (GAMMA) parameter
    fprintf(fModel, "KERNEL_GAMMA_PARAM: %f\n", (*param).KERNEL_GAMMA_PARAM);

    // kernel degree/slope parameter
    fprintf(fModel, "KERNEL_DEGREE_PARAM: %f\n", (*param).KERNEL_DEGREE_PARAM);

    // kernel coef/intercept parameter
    fprintf(fModel, "KERNEL_COEF_PARAM: %f\n", (*param).KERNEL_COEF_PARAM);

    // save the model
    fprintf(fModel, "MODEL:\n");
    for (i = 0; i < (*modelLLSVMlandmarks).size(); i++)
    {
        // put the i-th value of linear SVM hyperplane
        fprintf(fModel, "%2.6f", (double)modelLLSVMweightVector(i));

        // next, put the values of one row of modelLLSVMmatrixW
        for (j = 0; j < (*param).BUDGET_SIZE; j++)
            fprintf(fModel, " %2.6f", modelLLSVMmatrixW(i, j));

        // finally, store the landmark point
        for (j = 0; j < (*param).DIMENSION; j++)
        {
            if ((*((*modelLLSVMlandmarks)[i]))[j] != 0.0)
                fprintf(fModel, " %d:%2.6f", j + 1, (*((*modelLLSVMlandmarks)[i]))[j]);
        }
        fprintf(fModel, "\n");
    }

    fclose(fModel);
    return true;
}
```

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

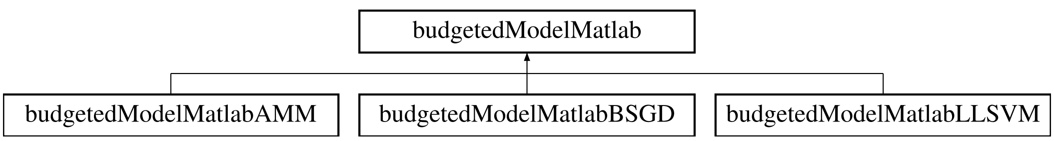
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/llsvm.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/llsvm.cpp

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 (mxArray *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	*matlabStruct*	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.

```
{
    if (mxGetNumberOfFields(matlabStruct) != NUM_OF_RETURN_FIELD)
        return -1;

    // get algorithm
    return (int)(*(mxGetPr(mxGetFieldByNumber(matlabStruct, 0, 0))));
}
```

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	*matlabStruct*	Pointer to Matlab structure.
in	*yLabels*	Vector of possible labels.
in	*param*	The parameters of the algorithm.
out	*msg*	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) x *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) x *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*) x *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*) x *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	*plhs*	Pointer to Matlab output.
in	*yLabels*	Vector of possible labels.
in	*param*	The parameters of the algorithm.

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) x *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) x *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*) x *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*) x *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:

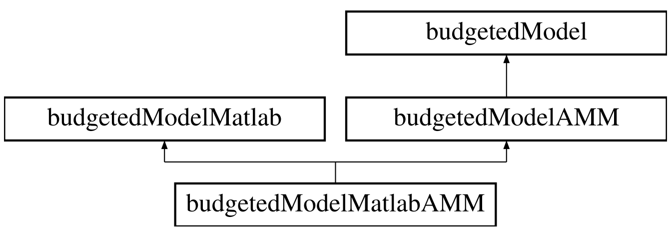
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.cpp

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	*matlabStruct*	Pointer to Matlab structure.
in	*yLabels*	Vector of possible labels.
in	*param*	The parameters of the algorithm.
out	*msg*	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 (("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 425 of file budgetedSVM_matlab.cpp.

```
{
    int i, j, numOfFields, numClasses, currClass, classCounter;
    double *ptr;
    int id = 0;
    mxArray **rhs;
    vector <unsigned int> numWeights;
    double sqrNorm;

    numOfFields = mxGetNumberOfFields(matlabStruct);
    if (numOfFields != NUM_OF_RETURN_FIELD)
    {
        *msg = "number of return fields is not correct";
        return false;
    }
    rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * numOfFields);

    for (i = 0; i < numOfFields; i++)
        rhs[i] = mxGetFieldByNumber(matlabStruct, 0, i);

    // algorithm
    ptr = mxGetPr(rhs[id]);
    param->ALGORITHM = (unsigned int)ptr[0];
    id++;

    // dimension
    ptr = mxGetPr(rhs[id]);
    param->DIMENSION = (unsigned int)ptr[0];
    id++;

    // numClasses
    ptr = mxGetPr(rhs[id]);
    numClasses = (unsigned int)ptr[0];
    id++;

    // labels
    if (mxIsEmpty(rhs[id]) == 0)
    {
        ptr = mxGetPr(rhs[id]);
        for(i = 0; i < numClasses; i++)
        {
            (*yLabels).push_back((int)ptr[i]);

            // add to model empty weight vector for each class
            vector <budgetedVectorAMM*> tempV;
            (*modelMM).push_back(tempV);
        }
    }
    id++;

    // numWeights
    if (mxIsEmpty(rhs[id]) == 0)
    {
        ptr = mxGetPr(rhs[id]);
        for(i = 0; i < numClasses; i++)
        {
            numWeights.push_back((int)ptr[i]);
        }
    }
    id++;

    // bias term
    ptr = mxGetPr(rhs[id]);
    param->BIAS_TERM = (double)ptr[0];
    id++;

    // kernel choice, just skip
    id++;

    // kernel width gamma
    id++;

    // kernel degree/slope param
    id++;

    // kernel intercept param
    id++;
```

```

    // weights
    int sr, sc;
    mwIndex *ir, *jc;

    sr = (int)mxGetN(rhs[id]);
    sc = (int)mxGetM(rhs[id]);

    ptr = mxGetPr(rhs[id]);
    ir = mxGetIr(rhs[id]);
    jc = mxGetJc(rhs[id]);

    // weights are in columns
    currClass = classCounter = 0;
    for (i = 0; i < sr; i++)
    {
        int low = (int)jc[i], high = (int)jc[i + 1];
        budgetedVectorAMM *eNew = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
        sqrNorm = 0.0;

        for (j = low; j < high; j++)
        {
            if (param->ALGORITHM == PEGASOS)
                ((*eNew)[(int)ir[j]]) = (float)ptr[j];
            else if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
            {
                if (j == low)
                    eNew->setDegradation(ptr[j]);
                else
                {
                    ((*eNew)[(int)ir[j] - 1]) = (float)ptr[j];
                    sqrNorm += (ptr[j] * ptr[j]);
                }
            }
        }
        eNew->setSqrL2norm(sqrNorm);
        (*modelMM)[currClass].push_back(eNew);
        eNew = NULL;

        // increment weight counter and check if new class is starting
        if (++classCounter == numWeights[currClass])
        {
            classCounter = 0;
            currClass++;
        }
    }
    id++;

    mxFree(rhs);
    return true;
}
```

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	*plhs*	Pointer to Matlab output.
in	*yLabels*	Vector of possible labels.
in	*param*	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.

```
{
    unsigned int i, j, numWeights = 0, cnt;
    double *ptr;
    mxArray *returnModel, **rhs;
    int outID = 0;

    rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * NUM_OF_RETURN_FIELD);

    // algorithm type
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->ALGORITHM;
    outID++;

    // dimension
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->DIMENSION;
    outID++;

    // number of classes
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = (double) (*yLabels).size();
    outID++;

    // labels
    rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    for (i = 0; i < (*yLabels).size(); i++)
        ptr[i] = (*yLabels)[i];
    outID++;

    // total number of weights
    rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    for (i = 0; i < (*modelMM).size(); i++)
    {
        ptr[i] = (double) (*modelMM)[i].size();
        numWeights += (unsigned int) (*modelMM)[i].size();
    }
    outID++;

    // bias param
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->BIAS_TERM;
    outID++;

    // kernel choice
    rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
    outID++;

    // kernel width gamma
    rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
    outID++;

    // kernel degree/slope param
    rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
    outID++;

    // kernel intercept param
    rhs[outID] = mxCreateDoubleMatrix(0, 0, mxREAL);
    outID++;

    // weights
    int irIndex, nonZeroElement;
    mwIndex *ir, *jc;

    // find how many non-zero elements there are
    nonZeroElement = 0;
    for (i = 0; i < (*modelMM).size(); i++)
    {
```

```
        for (j = 0; j < (*modelMM)[i].size(); j++)
        {
            for (unsigned int k = 0; k < (*param).DIMENSION; k++)             // for every feature
            {
                if ((*((*modelMM)[i][j]))[k] != 0.0)
                    nonZeroElement++;
            }
        }
    }

    // +1 is for degradation of AMM algorithms, it will be the first number in the row representing a weight
    if (param->ALGORITHM == PEGASOS)
        rhs[outID] = mxCreateSparse(param->DIMENSION, numWeights, nonZeroElement, mxREAL);
    else if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
        rhs[outID] = mxCreateSparse(param->DIMENSION + 1, numWeights, nonZeroElement + numWeights, mxREAL);
    ir = mxGetIr(rhs[outID]);
    jc = mxGetJc(rhs[outID]);
    ptr = mxGetPr(rhs[outID]);
    jc[0] = irIndex = cnt = 0;
    for (i = 0; i < (*modelMM).size(); i++)
    {
        for (j = 0; j < (*modelMM)[i].size(); j++)
        {
            int xIndex = 0;

            // this adds degradation to the beginning of a vector, more compact
            if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
            {
                ir[irIndex] = 0;
                ptr[irIndex] = (*modelMM)[i][j]->getDegradation();
                irIndex++, xIndex++;
            }

            // add the actual features
            for (unsigned int k = 0; k < (*param).DIMENSION; k++)             // for every feature
            {
                if ((*((*modelMM)[i][j]))[k] != 0.0)
                {
                    if ((param->ALGORITHM == AMM_BATCH) || (param->ALGORITHM == AMM_ONLINE))
                        ir[irIndex] = k + 1;
                    else if (param->ALGORITHM == PEGASOS)
                        ir[irIndex] = k;
                    ptr[irIndex] = (*((*modelMM)[i][j]))[k];
                    irIndex++, xIndex++;
                }
            }
            jc[cnt + 1] = jc[cnt] + xIndex;
            cnt++;
        }
    }
    // commented, since now it is appended to the weight matrix
    /*// degradations
    cnt = 0;
    rhs[outID] = mxCreateDoubleMatrix(numWeights, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    for (i = 0; i < (*modelMM).size(); i++)
        for (j = 0; j < (*modelMM)[i].size(); j++)
            ptr[cnt++] = (*modelMM)[i][j]->degradation;
    outID++;*/

    /* Create a struct matrix contains NUM_OF_RETURN_FIELD fields */
    returnModel = mxCreateStructMatrix(1, 1, NUM_OF_RETURN_FIELD, fieldNames);

    /* Fill struct matrix with input arguments */
    for(i = 0; i < NUM_OF_RETURN_FIELD; i++)
        mxSetField(returnModel, 0, fieldNames[i], mxDuplicateArray(rhs[i]));

    plhs[0] = returnModel;
    mxFree(rhs);
}
```

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

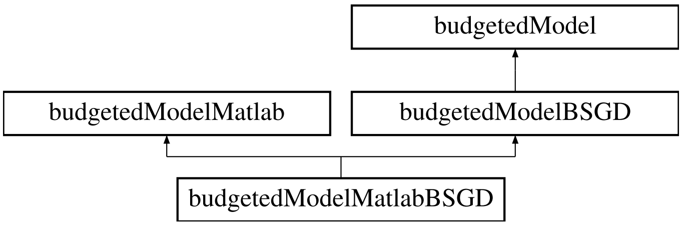
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.cpp

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	*matlabStruct*	Pointer to Matlab structure.
in	*yLabels*	Vector of possible labels.
in	*param*	The parameters of the algorithm.
out	*msg*	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") 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 708 of file budgetedSVM_matlab.cpp.

```
{
    int i, j, numOfFields, numClasses, currClass, classCounter;
    double *ptr;
    int id = 0;
    mxArray **rhs;
    vector <unsigned int> numWeights;
    double sqrNorm;

    numOfFields = mxGetNumberOfFields(matlabStruct);
    if (numOfFields != NUM_OF_RETURN_FIELD)
    {
        *msg = "number of return fields is not correct";
        return false;
    }
    rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * numOfFields);

    for (i = 0; i < numOfFields; i++)
        rhs[i] = mxGetFieldByNumber(matlabStruct, 0, i);

    // algorithm
    ptr = mxGetPr(rhs[id]);
    param->ALGORITHM = (unsigned int)ptr[0];
    id++;

    // dimension
    ptr = mxGetPr(rhs[id]);
    param->DIMENSION = (unsigned int)ptr[0];
    id++;

    // numClasses
    ptr = mxGetPr(rhs[id]);
    numClasses = (unsigned int)ptr[0];
    id++;

    // labels
    if (mxIsEmpty(rhs[id]) == 0)
    {
        ptr = mxGetPr(rhs[id]);
        for(i = 0; i < numClasses; i++)
            (*yLabels).push_back((int)ptr[i]);
    }
    id++;

    // numWeights, just skip
    id++;

    // bias term
    ptr = mxGetPr(rhs[id]);
    param->BIAS_TERM = (double)ptr[0];
    id++;

```

```
    // kernel choice, just skip
    ptr = mxGetPr(rhs[id]);
    param->KERNEL = (unsigned int) ptr[0];
    id++;

    // kernel width gamma
    ptr = mxGetPr(rhs[id]);
    param->KERNEL_GAMMA_PARAM = (double)ptr[0];
    id++;

    // kernel degree/slope param
    ptr = mxGetPr(rhs[id]);
    param->KERNEL_DEGREE_PARAM = (double)ptr[0];
    id++;

    // kernel intercept param
    ptr = mxGetPr(rhs[id]);
    param->KERNEL_COEF_PARAM = (double)ptr[0];
    id++;

    // weights
    int sr, sc;
    mwIndex *ir, *jc;

    sr = (int)mxGetN(rhs[id]);
    sc = (int)mxGetM(rhs[id]);

    ptr = mxGetPr(rhs[id]);
    ir = mxGetIr(rhs[id]);
    jc = mxGetJc(rhs[id]);

    // weights are in columns
    currClass = classCounter = 0;
    for (i = 0; i < sr; i++)
    {
        int low = (int)jc[i], high = (int)jc[i + 1];
        budgetedVectorBSGD *eNew = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT,
   numClasses);
        sqrNorm = 0.0;

        for (j = low; j < high; j++)
        {
            if ((unsigned int)ir[j] < (*yLabels).size())
            {
                // get alpha values
                eNew->alphas[(int)ir[j]] = ptr[j];
            }
            else
            {
                // get features
                ((*eNew)[(int)ir[j] - (int) (*yLabels).size()]) = (float)ptr[j];
                sqrNorm += (ptr[j] * ptr[j]);
            }
        }
        eNew->setSqrL2norm(sqrNorm);
        (*modelBSGD).push_back(eNew);
        eNew = NULL;
    }
    id++;

    mxFree(rhs);
    return true;
}
```

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	*plhs*	Pointer to Matlab output.
in	*yLabels*	Vector of possible labels.
in	*param*	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.

```
{
    unsigned int i, j, numWeights = 0, cnt;
    double *ptr;
    mxArray *returnModel, **rhs;
    int outID = 0;

    rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * NUM_OF_RETURN_FIELD);

    // algorithm type
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->ALGORITHM;
    outID++;

    // dimension
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->DIMENSION;
    outID++;

    // number of classes
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = (double) (*yLabels).size();
    outID++;

    // labels
    rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    for (i = 0; i < (*yLabels).size(); i++)
        ptr[i] = (*yLabels)[i];
    outID++;

    // total number of weights
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = (double) (*modelBSGD).size();
    numWeights = (unsigned int) (*modelBSGD).size();
    outID++;

    // bias param
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->BIAS_TERM;
    outID++;

    // kernel choice
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL;
    outID++;

    // kernel width gamma
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL_GAMMA_PARAM;
    outID++;

    // kernel degree/slope param
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL_DEGREE_PARAM;
    outID++;

    // kernel intercept param
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL_COEF_PARAM;
    outID++;

    // weights, different for MM algorithms, BSGD and LLSVM
```

```
    int irIndex, nonZeroElement;
    mwIndex *ir, *jc;

    // find how many non-zero elements there are
    nonZeroElement = 0;
    for (i = 0; i < (*modelBSGD).size(); i++)
    {
        // count non-zero features
        for (j = 0; j < (*param).DIMENSION; j++)
        {
            if ((*((*modelBSGD)[i]))[j] != 0.0)
                nonZeroElement++;
        }

        // count non-zero alphas also
        for (j = 0; j < (*yLabels).size(); j++)
        {
            if ((*((*modelBSGD)[i])).alphas[j] != 0.0)
                nonZeroElement++;
        }
    }

    //  +(*yLabels).size() is for the alpha parameters of each BSGD weight
    rhs[outID] = mxCreateSparse(param->DIMENSION + (*yLabels).size(), numWeights, nonZeroElement, mxREAL);
    ir = mxGetIr(rhs[outID]);
    jc = mxGetJc(rhs[outID]);
    ptr = mxGetPr(rhs[outID]);
    jc[0] = irIndex = cnt = 0;
    for (i = 0; i < (*modelBSGD).size(); i++)
    {
        int xIndex = 0;

        // this adds alpha weights to the beginning of a vector, more compact
        for (j = 0; j < (*yLabels).size(); j++)
        {
            if ((*((*modelBSGD)[i])).alphas[j] != 0.0)
            {
                ir[irIndex] = j;
                ptr[irIndex] = (*((*modelBSGD)[i])).alphas[j];
                irIndex++, xIndex++;
            }
        }

        // add the actual features
        for (j = 0; j < (*param).DIMENSION; j++)              // for every feature
        {
            if ((*((*modelBSGD)[i]))[j] != 0.0)
            {
                ir[irIndex] = j + (*yLabels).size();        // shift it to accomodate alpha weights
                ptr[irIndex] = (*((*modelBSGD)[i]))[j];
                irIndex++, xIndex++;
            }
        }
        jc[cnt + 1] = jc[cnt] + xIndex;
        cnt++;
    }

    /* Create a struct matrix contains NUM_OF_RETURN_FIELD fields */
    returnModel = mxCreateStructMatrix(1, 1, NUM_OF_RETURN_FIELD, fieldNames);

    /* Fill struct matrix with input arguments */
    for(i = 0; i < NUM_OF_RETURN_FIELD; i++)
        mxSetField(returnModel, 0, fieldNames[i], mxDuplicateArray(rhs[i]));

    plhs[0] = returnModel;
    mxFree(rhs);
}
```

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

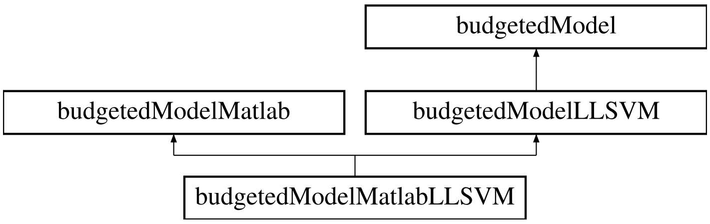
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.cpp

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 (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

5.10.1 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	*matlabStruct*	Pointer to Matlab structure.
in	*yLabels*	Vector of possible labels.
in	*param*	The parameters of the algorithm.
out	*msg*	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.

```
{
    unsigned int i, j, numOfFields, numClasses;
    double *ptr, sqrNorm;
    int id = 0;
    mxArray **rhs;

    numOfFields = mxGetNumberOfFields(matlabStruct);
    if (numOfFields != NUM_OF_RETURN_FIELD)
    {
        *msg = "Number of return fields is not correct.";
        return false;
    }
    rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * numOfFields);

    for (i = 0; i < numOfFields; i++)
        rhs[i] = mxGetFieldByNumber(matlabStruct, 0, i);

    // algorithm
    ptr = mxGetPr(rhs[id]);
    param->ALGORITHM = (unsigned int)ptr[0];
    id++;

    // dimension
    ptr = mxGetPr(rhs[id]);
    param->DIMENSION = (unsigned int)ptr[0];
    id++;

    // numClasses
    ptr = mxGetPr(rhs[id]);
    numClasses = (unsigned int)ptr[0];
    id++;

    // labels
    if (mxIsEmpty(rhs[id]) == 0)
    {
        ptr = mxGetPr(rhs[id]);
        for(i = 0; i < numClasses; i++)
            (*yLabels).push_back((int)ptr[i]);
    }
    id++;

    // numWeights
    ptr = mxGetPr(rhs[id]);
    param->BUDGET_SIZE = (unsigned int) ptr[0];
    id++;

    // bias term
    ptr = mxGetPr(rhs[id]);
    param->BIAS_TERM = (double) ptr[0];
    id++;

    // kernel choice
    ptr = mxGetPr(rhs[id]);
    param->KERNEL = (unsigned int) ptr[0];
    id++;

    // kernel width gamma
    ptr = mxGetPr(rhs[id]);
    param->KERNEL_GAMMA_PARAM = (double) ptr[0];
    id++;

    // kernel degree/slope param
    ptr = mxGetPr(rhs[id]);
    param->KERNEL_DEGREE_PARAM = (double) ptr[0];
    id++;
```

```

    // kernel intercept param
    ptr = mxGetPr(rhs[id]);
    param->KERNEL_COEF_PARAM = (double) ptr[0];
    id++;

    // weights
    unsigned int sr, sc;
    mwIndex *ir, *jc;

    sr = (int)mxGetN(rhs[id]);
    sc = (int)mxGetM(rhs[id]);

    ptr = mxGetPr(rhs[id]);
    ir = mxGetIr(rhs[id]);
    jc = mxGetJc(rhs[id]);

    // allocate memory for model
    modelLLSVMmatrixW.resize((*param).BUDGET_SIZE, (*param).BUDGET_SIZE);
    modelLLSVMweightVector.resize((*param).BUDGET_SIZE, 1);

    // weight-vectors are in columns
    for (i = 0; i < sr; i++)
    {
        unsigned int low = (int)jc[i], high = (int)jc[i + 1];
        budgetedVectorLLSVM *eNew = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
        sqrNorm = 0.0;

        // get the linear weight
        modelLLSVMweightVector(i, 0) = ptr[low];

        // get the modelLLSVMmatrixW
        for (j = low + 1; j < low + (*param).BUDGET_SIZE + 1; j++)
            modelLLSVMmatrixW(i, j - low - 1) = ptr[j];

        // get the features
        for (j = low + (*param).BUDGET_SIZE + 1; j < high; j++)
        {
            ((*eNew)[(int)ir[j] - (*param).BUDGET_SIZE - 1]) = (float)ptr[j];
            sqrNorm += (ptr[j] * ptr[j]);
        }
        eNew->setSqrL2norm(sqrNorm);
        (*modelLLSVMlandmarks).push_back(eNew);
        eNew = NULL;
    }
    id++;

    mxFree(rhs);
    return true;
}
```

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	*plhs*	Pointer to Matlab output.
in	*yLabels*	Vector of possible labels.
in	*param*	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.

```
{
    unsigned int i, j, numWeights = 0, cnt;
    double *ptr;
    mxArray *returnModel, **rhs;
    int outID = 0;

    rhs = (mxArray **) mxMalloc(sizeof(mxArray *) * NUM_OF_RETURN_FIELD);

    // algorithm type
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->ALGORITHM;
    outID++;

    // dimension
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->DIMENSION;
    outID++;

    // number of classes
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = (double) (*yLabels).size();
    outID++;

    // labels
    rhs[outID] = mxCreateDoubleMatrix((*yLabels).size(), 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    for (i = 0; i < (*yLabels).size(); i++)
        ptr[i] = (*yLabels)[i];
    outID++;

    // total number of weights
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = (double) (*modelLLSVMlandmarks).size();
    numWeights = (unsigned int) (*modelLLSVMlandmarks).size();
    outID++;

    // bias param
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->BIAS_TERM;
    outID++;

    // kernel choice
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL;
    outID++;

    // kernel width gammma
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL_GAMMA_PARAM;
    outID++;

    // kernel degree/slope param
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL_DEGREE_PARAM;
    outID++;

    // kernel intercept param
    rhs[outID] = mxCreateDoubleMatrix(1, 1, mxREAL);
    ptr = mxGetPr(rhs[outID]);
    ptr[0] = param->KERNEL_COEF_PARAM;
    outID++;

    // weights, different for MM algorithms, BSGD and LLSVM
    int irIndex, nonZeroElement;
    mwIndex *ir, *jc;

    // find how many non-zero elements there are
    nonZeroElement = 0;
    for (i = 0; i < (*modelLLSVMlandmarks).size(); i++)
```

```
    {
        // count non-zero features
        for (j = 0; j < (*param).DIMENSION; j++)
        {
            if ((*((*modelLLSVMlandmarks)[i]))[j] != 0.0)
                nonZeroElement++;
        }

        // count all elements of modelLLSVMmatrixW also
        nonZeroElement += (numWeights * numWeights);

        // count linear SVM length also
        nonZeroElement += numWeights;
    }

    //  +(*yLabels).size() is for the alpha parameters of each BSGD weight
    rhs[outID] = mxCreateSparse(param->DIMENSION + numWeights + 1, numWeights, nonZeroElement, mxREAL);
    ir = mxGetIr(rhs[outID]);
    jc = mxGetJc(rhs[outID]);
    ptr = mxGetPr(rhs[outID]);
    jc[0] = irIndex = cnt = 0;
    for (i = 0; i < (*modelLLSVMlandmarks).size(); i++)
    {
        int xIndex = 0;

        // this adds alpha weights to the beginning of a vector, more compact
        ir[irIndex] = 0;
        ptr[irIndex] = modelLLSVMweightVector(i, 0);
        irIndex++, xIndex++;

        // this adds row of modelLLSVMmatrixW next, more compact
        for (j = 0; j < numWeights; j++)
        {
            ir[irIndex] = j + 1;        // shift it to accomodate linear weight
            ptr[irIndex] = modelLLSVMmatrixW(i, j);
            irIndex++, xIndex++;
        }

        // add the actual features
        for (j = 0; j < (*param).DIMENSION; j++)
        {
            if ((*((*modelLLSVMlandmarks)[i]))[j] != 0.0)
            {
                ir[irIndex] = j + numWeights + 1;       // shift it to accomodate linear weight and
       modelLLSVMmatrixW
                ptr[irIndex] = (*((*modelLLSVMlandmarks)[i]))[j];
                irIndex++, xIndex++;
            }
        }
        jc[cnt + 1] = jc[cnt] + xIndex;
        cnt++;
    }

    /* Create a struct matrix contains NUM_OF_RETURN_FIELD fields */
    returnModel = mxCreateStructMatrix(1, 1, NUM_OF_RETURN_FIELD, fieldNames);

    /* Fill struct matrix with input arguments */
    for(i = 0; i < NUM_OF_RETURN_FIELD; i++)
        mxSetField(returnModel, 0, fieldNames[i], mxDuplicateArray(rhs[i]));

    plhs[0] = returnModel;
    mxFree(rhs);
}
```

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

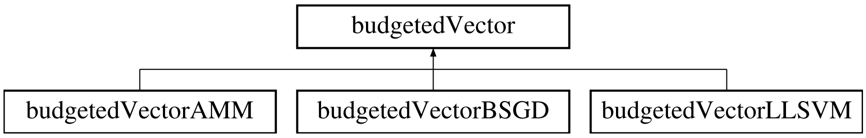
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/matlab/budgetedSVM_matlab.cpp

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	*dim*	Dimensionality of the vector.
in	*chnkWght*	Size of each vector chunk.

Definition at line 651 of file budgetedSVM.h.

```
    {
        dimension = dim;
        arrayLength = 0;
        if (chnkWght > 0)
            chunkWeight = chnkWght;
        else
            svmPrintErrorString("In budgetedVector(), weight chunk must be positive integer!\n");

        if (dim != 0)
            arrayLength = (unsigned int)((dim - 1) / chunkWeight) + 1;

        // just initialize the elements of array to NULL, will be created only
        //      when needed, when one of the elements becomes non-zero
        for (unsigned int i = 0; i < arrayLength; i++)
            array.push_back(NULL);

        weightID = id++;
        sqrL2norm = 0.0;
    }
```

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	*otherVector*	The second input vector to kernel.
in	*param*	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.

```
{
    switch ((*param).KERNEL)
    {
        case KERNEL_FUNC_GAUSSIAN:
            return gaussianKernel(otherVector, param);
            break;

        case KERNEL_FUNC_EXPONENTIAL:
            return exponentialKernel(otherVector, param);
            break;

        case KERNEL_FUNC_SIGMOID:
            return sigmoidKernel(otherVector, param);
            break;

        case KERNEL_FUNC_POLYNOMIAL:
            return polyKernel(otherVector, param);
            break;

        case KERNEL_FUNC_LINEAR:
            return linearKernel(otherVector);
            break;

        case KERNEL_FUNC_USER_DEFINED:
            return userDefinedKernel(otherVector, param);
            break;

        default:
            svmPrintErrorString("Error, undefined kernel function found! Run 'budgetedsvm-train' for help.\n");
            return -1.0;
    }
}
```

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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*param*	The parameters of the algorithm.
in	*inputVectorSqrNorm*	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.

```
{
    switch ((*param).KERNEL)
    {
        case KERNEL_FUNC_GAUSSIAN:
            return gaussianKernel(t, inputData, param, inputVectorSqrNorm);
            break;

        case KERNEL_FUNC_EXPONENTIAL:
            return exponentialKernel(t, inputData, param, inputVectorSqrNorm);
            break;

        case KERNEL_FUNC_SIGMOID:
            return sigmoidKernel(t, inputData, param);
            break;

        case KERNEL_FUNC_POLYNOMIAL:
            return polyKernel(t, inputData, param);
            break;

        case KERNEL_FUNC_LINEAR:
            return linearKernel(t, inputData, param);
            break;

        case KERNEL_FUNC_USER_DEFINED:
            return userDefinedKernel(t, inputData, param);
            break;

        default:
            svmPrintErrorString("Error, undefined kernel function found! Run 'budgetedsvm-train' for
  help.\n");
            return -1.0;
    }
}
```

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	*inputData*	Input data from which t-th vector is considered.
in	*t*	Index of the input vector in the input data.
in	*param*	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.

```
{
    unsigned int ibegin = inputData->ai[t];
    unsigned int iend = (t == (unsigned int) (inputData->ai.size() - 1)) ? (unsigned int)
   (inputData->aj.size()) : inputData->ai[t + 1];

    this->clear();
    for (unsigned int i = ibegin; i < iend; i++)
    {
        ((*this)[inputData->aj[i] - 1]) = inputData->an[i];
        sqrL2norm += (inputData->an[i] * inputData->an[i]);
    }

    if ((*param).BIAS_TERM != 0)
    {
        ((*this)[(*param).DIMENSION - 1]) = (float)((long double)(*param).BIAS_TERM);
        sqrL2norm += ((*param).BIAS_TERM * (*param).BIAS_TERM);
    }
};
```

5.11.3.4 createVectorUsingVector()

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

Create new vector from the existing one.

Parameters

in	*existingVector*	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.

```
{
    clear();
    for (unsigned int i = 0; i < arrayLength; i++)
    {
        if (existingVector->array[i] != NULL)
        {
            array[i] = new (nothrow) float[chunkWeight];
            for (unsigned int j = 0; j < chunkWeight; j++)
                array[i][j] = existingVector->array[i][j];
        }
    }
    sqrL2norm = existingVector->sqrL2norm;
}
```

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	*otherVector*	The second input vector to exponential kernel.
in	*param*	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.

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

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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*param*	The parameters of the algorithm.
in	*inputVectorSqrNorm*	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.

```
{
    if (inputVectorSqrNorm == 0.0)
        inputVectorSqrNorm = inputData->getVectorSqrL2Norm(t, param);

    long double temp = sqrt((long double) (this->sqrL2norm + inputVectorSqrNorm - 2.0L * this->linearKernel(t, inputData, param)));
    if (temp >= 0)
        return exp(-0.5L * (long double)((*param).KERNEL_GAMMA_PARAM) * temp);
    else
        return 0.0L;
}
```

5.11.3.7 extendDimensionality()

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

Extend the dimensionality of the vector.

Parameters

in	*newDim*	New dimensionality of the vector.
in	*param*	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 variaty 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.

```
{
    if (dimension > newDim)
    {
        svmPrintErrorString("In extendDimensionality(), extended vector dimensionality smaller than the old one!\n");
    }
    else
    {
        /*char text[127];
        sprintf(text, "In the func, current: %d\tnew: %d!\n", dimension, newDim);
        svmPrintString(text);*/
    }

    // when extending the vector, only the last element of the chunk array is modified,
    //  and possibly more zero-chunks are added after the last array element
    unsigned int newArrayLength = (unsigned int)((newDim - 1) / chunkWeight) + 1;

    float biasTerm = 0.0;
    if (param->BIAS_TERM != 0.0)
    {
        biasTerm = (*this)[dimension - 1];
    }

    unsigned int lastElementLength = dimension % chunkWeight;
    if (lastElementLength == 0)
        lastElementLength = chunkWeight;

    unsigned int newLastElementLength = newDim % chunkWeight;
    if (newLastElementLength == 0)
        newLastElementLength = chunkWeight;

    float *temp = NULL;
    if (newArrayLength == arrayLength)
```

```
    {
        // 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
        // if the new and the old array lengths are the same, then possibly the new chunk element is also smaller than chunkWeight
        temp = new float[newLastElementLength];
        for (unsigned int i = 0; i < newLastElementLength; i++)
        {
            if (i < (lastElementLength - (int)(param->BIAS_TERM != 0.0)))  // -1 to not copy the bias term
            {
                temp[i] = array[arrayLength - 1][i];     // copy the entire last element of chunk-array
            }
            else
            {
                temp[i] = 0.0;                           // set the remaining elements to zero
            }
        }
    }
    else if (newArrayLength > arrayLength)
    {
        // in this case, pad the rest of the current last element with zeros, and new NULL weights will be created
        temp = new float[chunkWeight];
        for (unsigned int i = 0; i < chunkWeight; i++)
        {
            if (i < (lastElementLength - (int)(param->BIAS_TERM != 0.0)))  // -1 to not copy the bias term
            {
                temp[i] = array[arrayLength - 1][i];     // copy the entire last element of chunk-array
            }
            else
            {
                temp[i] = 0.0;                           // set the remaining elements to zero
            }
        }

        // initialize the additional elements of array to NULL
        for (unsigned int i = 0; i < newArrayLength - arrayLength; i++)
            array.push_back(NULL);
    }
    else
    {
        // just a sanity check
        svmPrintErrorString("Error in extendDimensionality(): New array length shorter than old one, should never happen!");
    }

    // put the new, longer chunk instead of the old one
    delete [] array[arrayLength - 1];
    array[arrayLength - 1] = temp;
    temp = NULL;

    // set the static parameters of the budgetedVector class to new values
    arrayLength = newArrayLength;
    dimension = newDim;

    // put the bias term to the end if it exists
    if (param->BIAS_TERM != 0.0)
    {
        (*this)[dimension - 1] = biasTerm;
    }
}
```

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	*otherVector*	The second input vector to RBF kernel.
in	*param*	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.

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

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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*inputVectorSqrNorm*	If zero or not provided, the norm of t-th vector from inputData is computed on-the-fly.
in	*param*	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.

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

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.

```
{
    return dimension;
}
```

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.

```
{
    return weightID;
}
```

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.

```
{
    return sqrL2norm;
}
```

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	*otherVector*	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.

```
{
    long double result = 0.0L;
    unsigned long chunkSize = chunkWeight;
    for (unsigned int i = 0; i < arrayLength; i++)
    {
        // if either of them is NULL, meaning all-zeros vector chunk, move on to the next chunk
        if ((this->array[i] == NULL) || (otherVector->array[i] == NULL))
            continue;

        // now we know that i-th vector chunks of both vectors have non-zero elements, go one by one and compute linear kernel
        if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
            chunkSize = dimension % chunkWeight;
        for (unsigned int j = 0; j < chunkSize; j++)
        {
            result += this->array[i][j] * otherVector->array[i][j];
        }
    }
    return result;
}
```

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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*param*	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.

```
{
    long double result = 0.0;
    long unsigned int pointIndexPointer = inputData->ai[t];
    long unsigned int maxPointIndex = ((unsigned int)(t + 1) == inputData->N) ? inputData->aj.size() : inputData->ai[t + 1];
    char text[256];
    unsigned int idx, vectorInd, arrayInd;

    for (long unsigned int i = pointIndexPointer; i < maxPointIndex; i++)
    {
        idx = inputData->aj[i] - 1;
        vectorInd = (int) (idx / chunkWeight);
        arrayInd = (int) (idx % chunkWeight);

        // if the input vector is longer than the budgeted vector, this can happen when the test data has
        //  vectors with dimensionality that is longer than previously seen during training, check your test data!
        if (vectorInd >= arrayLength)
        {
            sprintf(text, "Error, input vector is longer than the budgeted vector, detected dimension %d in function linearKernel(), check your input data.\n", idx + 1);
            svmPrintErrorString(text);
        }

        // this means that all elements of this chunk are 0
        if (array[vectorInd] == NULL)
            continue;
        else
            result += array[vectorInd][arrayInd] * inputData->an[i];
    }
    if ((*param).BIAS_TERM != 0)
        result += (((*this)[(*param).DIMENSION - 1]) * (*param).BIAS_TERM);
    return result;
}
```

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.

```
{
    unsigned int vectorInd = (unsigned int)(idx / (int) chunkWeight);
    unsigned int arrayInd = (unsigned int) (idx % (int) chunkWeight);

    // if the input vector is longer than the budgeted vector, this can happen when the test data has
    //  vectors with dimensionality that is longer than previously seen during training, check your test data!
    if (vectorInd >= arrayLength)
    {
        svmPrintErrorString("Error, input vector is longer than the budgeted vector in function budgetedVector::operator[], check your input data.\n");
    }

    // if all elements were zero, then first create the array and only
    //    then return the reference
    if (array[vectorInd] == NULL)
    {
```

```
        float *tempArray = NULL;
        unsigned long arraySize = chunkWeight;

        // if the last chunk, then it might be smaller than the rest
        if (vectorInd == (arrayLength - 1))
        {
            arraySize = dimension % chunkWeight;
            if (arraySize == 0)
                arraySize = chunkWeight;
            tempArray = new (nothrow) float[arraySize];
        }
        else
            tempArray = new (nothrow) float[chunkWeight];

        if (tempArray == NULL)
        {
            svmPrintErrorString("Memory allocation error (budgetedVector assignment)!");
        }

        // null the array
        for (unsigned int j = 0; j < arraySize; j++)
            *(tempArray + j) = 0;

        array[vectorInd] = tempArray;
    }

    return *(array[vectorInd] + arrayInd);
}
```

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	*idx*	Index of vector element that is retrieved.

Returns

Value of the element of the vector.

Definition at line 639 of file budgetedSVM.cpp.

```
{
    unsigned int vectorInd = (unsigned int) (idx / (int) chunkWeight);
    unsigned int arrayInd = (unsigned int) (idx % (int) chunkWeight);

    // this means that all elements of this chunk are 0
    if (array[vectorInd] == NULL)
        return 0.0;
    else
        return *(array[vectorInd] + arrayInd);
}
```

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	*otherVector*	The second input vector to polynomial kernel.
in	*param*	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.

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

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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*param*	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.

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

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	*otherVector*	The second input vector to sigmoid kernel.
in	*param*	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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*param*	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.

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

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.

```
{
    long double tempSum = 0.0;
    unsigned long chunkSize = chunkWeight;

    for (unsigned int i = 0; i < arrayLength; i++)
    {
        if (array[i] != NULL)
        {
            if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
                chunkSize = dimension % chunkWeight;

            for (unsigned int j = 0; j < chunkSize; j++)
                tempSum += ((long double)array[i][j] * (long double)array[i][j]);
        }
    }
    return tempSum;
}
```

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	*otherVector*	The second input vector to user-defined kernel.
in	*param*	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.

```
{
    // 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
    svmPrintString("\nError, non-implemented user-defined kernel function invoked!\n");
    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");
    return -1.0;
}
```

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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*param*	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.

```
{
    // 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
    svmPrintString("\nError, non-implemented user-defined kernel function!\n");
    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");
    return -1.0;
}
```

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 corresponing 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:

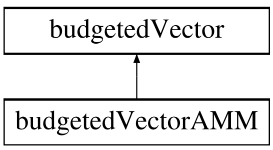
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/budgetedSVM.cpp

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	*dim*	Dimensionality of the vector.
in	*chnkWght*	Size of each vector chunk.

Definition at line 50 of file mm_algs.h.

```
: budgetedVector(dim,
chnkWght)
        {
            degradation = 1.0;
        }
```

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	*inputData*	Input data from which t-th vector is considered.
in	*oto*	Total number of iterations so far.
in	*t*	Index of the input vector in the input data.
in	*param*	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.

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

5.12.3.2 createVectorUsingVector()

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

Create new vector from the existing one.

Parameters

in	*existingVector*	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	*oto*	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

in	*otherVector*	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() *
       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	*t*	Index of the input vector in the input data.
in	*inputData*	Input data from which t-th vector is considered.
in	*param*	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	*iteration*	Training iteration at which the degradation is set, used to compute the degradation value.
in	*param*	The parameters of the algorithm.

Definition at line 106 of file mm_algs.h.

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

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

```
{
    unsigned long pointIndexPointer = inputData->ai[t];
    unsigned long maxPointIndex = ((t + 1) == (unsigned int) inputData->ai.size()) ? (unsigned int) inputData->aj.size() : inputData->ai[t + 1];

    long double linKern = this->linearKernel(t, inputData, param);
    long double divisor = (long double)sign * ((long double)oto + 1.0) * (long double)(*param).LAMBDA_PARAM * degradation;
    for (unsigned long i = pointIndexPointer; i < maxPointIndex; i++)
    {
        ((*this)[inputData->aj[i] - 1]) = (float)((long double)((*this)[inputData->aj[i] - 1]) + (long double)inputData->an[i] / divisor);
    }
    if ((*param).BIAS_TERM != 0)
    {
        ((*this)[(*param).DIMENSION - 1]) = (float)((long double)((*this)[(*param).DIMENSION - 1]) + (long double)(*param).BIAS_TERM / divisor);
    }

    this->sqrL2norm += (long double)inputData->getVectorSqrL2Norm(t, param) / (divisor * divisor) + 2.0L / (divisor * this->degradation) * linKern;
}
```

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	*otherVector*	Misclassified example used to update the existing weight.
in	*oto*	Total number of iterations so far.
in	*sign*	+1 if the input vector is of the true class, -1 otherwise, specifies how the weights will be updated.
in	*param*	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 missclassified example used to update an existing weight is located in the budgetedVectorAMM object.

Definition at line 308 of file mm_algs.cpp.

```
{
    unsigned long chunkSize = chunkWeight;
    unsigned int i, j;
    float *tempArray = NULL;
    long double divisor = (long double)sign * ((long double)oto + 1.0) * (long double)(*param).LAMBDA_PARAM * degradation;
    long double linKern = this->linearKernel(otherVector);
    for (i = 0; i < arrayLength; i++)
    {
        // if the input vector's i-th array is NULL, then there is no need to update any of this vector's features
        if (otherVector->array[i] == NULL)
            continue;

        // now we know that i-th vector chunk of input vector has non-zero elements, go one by one and this vector
        if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
            chunkSize = dimension % chunkWeight;

        // if the i-th chunk weight is NULL then create it
        if (this->array[i] == NULL)
        {
            // create and null the array
            tempArray = new (nothrow) float[chunkSize];
            for (j = 0; j < chunkSize; j++)
                *(tempArray + j) = 0;
            this->array[i] = tempArray;
        }
        else
            tempArray = this->array[i];

        for (j = 0; j < chunkSize; j++)
        {
            *(tempArray + j) += (float)((long double) otherVector->array[i][j] / divisor);
        }
        tempArray = NULL;
    }

    sqrL2norm += (long double)otherVector->getSqrL2norm() / (divisor * divisor) + 2.0L / (divisor * this->degradation) * linKern;
}
```

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:

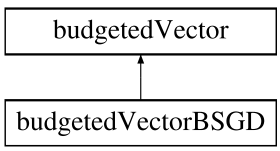
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/mm_algs.cpp

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

Definition at line 71 of file bsgd.h.

```
                                                                                                    :
      budgetedVector(dim, chnkWght)
        {
            if (numClasses == 0)
                numClasses = numCls;

            for (unsigned int i = 0; i < numClasses; i++)
                this->alphas.push_back(0.0);
        }
```

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.

```
{
    long double tempSum = 0.0;
    for (unsigned long i = 0; i < alphas.size(); i++)
        tempSum += (alphas[i] * alphas[i]);
    return tempSum;
}
```

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.

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

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.

```
        {
            return numClasses;
        }
```

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	*v*	Vector that is merged with this vector.
in	*kMax*	Parameter that specifies how to combine them (currentVector <- kMax ∗ 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.

```
{
    unsigned long chunkSize = chunkWeight;
```

```
    unsigned long i, j;
    long double linKern = this->linearKernel(v);

    for (i = 0; i < arrayLength; i++)
    {
        if (this->array[i] != NULL)
        {
            if ((*v).array[i] == NULL)
            {
                if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
                    chunkSize = dimension % chunkWeight;
                for (j = 0; j < chunkSize; j++)
                    array[i][j] = (float)(kMax * (long double) this->array[i][j]);
            }
            else
            {
                if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
                    chunkSize = dimension % chunkWeight;
                for (j = 0; j < chunkSize; j++)
                    this->array[i][j] = (float)(kMax * this->array[i][j] + (1.0 - kMax) *
   (*v).array[i][j]);
            }
        }
        else
        {
            if ((*v).array[i] != NULL)
            {
                if ((i == (arrayLength - 1)) && (dimension != chunkWeight))
                    chunkSize = dimension % chunkWeight;

                float *tempArray = new (nothrow) float[chunkSize];
                if (tempArray == NULL)
                {
                    svmPrintErrorString("Memory allocation error (budgetedVector assignment)!");
                }

                // copy the array
                for (j = 0; j < chunkSize; j++)
                    *(tempArray + j) = (float)((1.0 - kMax) * (*v).array[i][j]);

                this->array[i] = tempArray;
                tempArray = NULL;
            }
        }
    }

    // we also update the squared norm of the merged vector
    this->sqrL2norm = kMax * kMax * (long double) (this->sqrL2norm) + (1.0L - kMax) * (1.0L - kMax) *
   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:

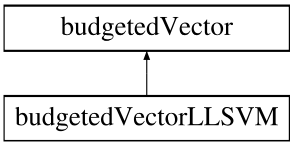
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.h
- C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/src/bsgd.cpp

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	*dataVector*	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.

```
        {
            for (unsigned int i = 0; i < (unsigned int) dataVector.size(); i++)
            {
                if (dataVector[i] != 0.0)
                {
                    (*this)[i] = (float) dataVector[i];
                    sqrL2norm += (dataVector[i] * dataVector[i]);
                }
            }
        };
```

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 ∗ KERNEL_GAMMA_PARAM ∗ ||x - y||^2) (default: 1/DIME↩ NSIONALITY)
- double KERNEL_DEGREE_PARAM

  Degree of polynomial kernel (x^T ∗ y + KERNEL_COEF_PARAM)^KERNEL_DEGREE_PARAM, OR slope parameter of sigmoid kernel tanh(KERNEL_DEGREE_PARAM ∗ x^T ∗ y + KERNEL_COEF_PARAM) (default: 2)
- double KERNEL_COEF_PARAM

  Coefficient of polynomial kernel (x^T ∗ y + KERNEL_COEF_PARAM)^KERNEL_DEGREE_PARAM, or intercept of sigmoid kernel tanh(KERNEL_DEGREE_PARAM ∗ x^T ∗ y + 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	*dataSparsity*	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::linearKern
budgetedVector::gaussianKernel(unsigned int, budgetedData∗, parameters∗, long double), budgetedVector::gaussianKernel(bu

Definition at line 314 of file budgetedSVM.h.

```
{
    // if the parameter is already set then just return and change nothing; it can be that it was set by a user
    //  or it was already set when the earlier data chunks were loaded
    if ((VERY_SPARSE_DATA == 0) || (VERY_SPARSE_DATA == 1))
        return;

    // if the sparsity is less than 5%, then we say that we are working with very sparse data
    if (dataSparsity < 5.0)
        VERY_SPARSE_DATA = 1;
    else
        VERY_SPARSE_DATA = 0;
};
```

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 * KERNEL_GAMMA_PARAM * ||x - y||^2)
- Exponential: K(x, y) = exp(-0.5 * KERNEL_GAMMA_PARAM * ||x - y||)
- Polynomial: K(x, y) = (x^T * y + KERNEL_COEF_PARAM)^KERNEL_DEGREE_PARAM
- Linear: K(x, y) = (x^T * y)
- Sigmoid: K(x, y) = tanh(KERNEL_DEGREE_PARAM * x^T * y + KERNEL_COEF_PARAM)
- User-defined: To add your kernel function please open file 'src/budgetedSVM.cpp' and modify two user↵ DefinedKernel() 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., 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_SAMPLIN↩G_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∗, l 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/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 ∗modelFileName=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	*plhs*	Pointer to Matlab output.

Definition at line 1118 of file budgetedSVM_matlab.cpp.

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

6.1.2.2 parseInputMatlab()

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

Parses the user input and modifies parameter settings as necessary.

Parameters

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

Definition at line 1249 of file budgetedSVM_matlab.cpp.

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

```
    vector <char> option;
    vector <float> value;
    FILE *pFile = NULL;

    if (paramString == NULL)
        len = 0;
    else
        len = (int) strlen(paramString);

    // check if the input data file exists only if input data filename is provided
    if (inputFileName)
    {
        if (!readableFileExists(inputFileName))
        {
            sprintf(str, "Can't open input file %s!\n", inputFileName);
            mexErrMsgTxt(str);
        }
    }

    while (pos < len)
    {
        if (paramString[pos++] == '-')
        {
            option.push_back(paramString[pos]);
            pos += 2;

            tempPos = 0;
            while ((paramString[pos] != ' ') && (paramString[pos] != '\0'))
            {
                str[tempPos++] = paramString[pos++];
            }
            str[tempPos++] = '\0';
            value.push_back((float) atof(str));
        }
    }

    if (trainingPhase)
    {
        // check if the model file exists only if model filename is provided
        if (modelFileName)
        {
            pFile = fopen(modelFileName, "w");
            if (pFile == NULL)
            {
                sprintf(str, "Can't create model file %s!\n", modelFileName);
                mexErrMsgTxt(str);
            }
            else
            {
                fclose(pFile);
                pFile = NULL;
            }
        }

        // modify parameters
        for (unsigned int i = 0; i < option.size(); i++)
        {
            switch (option[i])
            {
                case 'A':
                    (*param).ALGORITHM = (unsigned int) value[i];
                    if ((*param).ALGORITHM > 4)
                    {
                        sprintf(str, "Input parameter '-A %d' out of bounds!\nRun 'budgetedsvm_train()' for help.", (*param).ALGORITHM);
                        mexErrMsgTxt(str);
                    }
                    break;
                case 'e':
                    (*param).NUM_EPOCHS = (unsigned int) value[i];
                    break;
                case 's':
                    (*param).NUM_SUBEPOCHS = (unsigned int) value[i];
                    break;
                case 'k':
                    (*param).K_PARAM = (unsigned int) value[i];
                    break;
                case 'c':
                    (*param).C_PARAM = value[i];
                    if ((*param).C_PARAM <= 0.0)
                    {
                        sprintf(str, "Input parameter '-c' should be a positive real number!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;
                case 'L':
```

```
                    (*param).LAMBDA_PARAM = (double) value[i];
                    if ((*param).LAMBDA_PARAM <= 0.0)
                    {
                        sprintf(str, "Input parameter '-L' should be a positive real number!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;

                case 'K':
                    (*param).KERNEL = (unsigned int) value[i];
                    break;

                case 'g':
                    (*param).KERNEL_GAMMA_PARAM = (long double) value[i];
                    if ((*param).KERNEL_GAMMA_PARAM <= 0.0)
                    {
                        sprintf(str, "Input parameter '-g' should be a positive real number!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;

                case 'd':
                    (*param).KERNEL_DEGREE_PARAM = (double) value[i];
                    if ((*param).KERNEL_DEGREE_PARAM <= 0.0)
                    {
                        sprintf(str, "Input parameter '-d' should be a positive real number!\nRun 'budgetedsvm-train()' for help.\n");
                        mexErrMsgTxt(str);
                    }
                    break;

                case 'i':
                    (*param).KERNEL_COEF_PARAM = (double) value[i];
                    break;

                case 'm':
                    (*param).MAINTENANCE_SAMPLING_STRATEGY = (unsigned int) value[i];
                    break;

                case 'b':
                    (*param).BIAS_TERM = (double) value[i];
                    break;
                case 'v':
                    (*param).VERBOSE = (value[i] != 0);
                    break;
                case 'r':
                    (*param).RANDOMIZE = (value[i] != 0);
                    break;
                case 'B':
                    (*param).BUDGET_SIZE = (unsigned int) value[i];
                    if ((*param).BUDGET_SIZE < 1)
                    {
                        sprintf(str, "Input parameter '-B' should be a positive integer!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;
                case 'D':
                    // a user explicitly assigns dimensionality only if data set is given in .txt file, otherwise dimensionality is found directly from Matlab, no need for a user to specify it
                    if (inputFileName)
                    {
                        (*param).DIMENSION = (unsigned int) value[i];
                    }
                    else
                    {
                        //sprintf(str, "Warning, if data loaded to Matlab no need to set '-D %d' option.\nRun 'budgetedsvm_train()' for help.\n", (int) value[i]);
                        //mexPrintf(str);
                    }
                    break;

                case 'z':
                    (*param).CHUNK_SIZE = (unsigned int) value[i];
                    if ((*param).CHUNK_SIZE < 1)
                    {
                        sprintf(str, "Input parameter '-z' should be an integer larger than 0!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;
                case 'w':
                    (*param).CHUNK_WEIGHT = (unsigned int) value[i];
                    if ((*param).CHUNK_WEIGHT < 1)
                    {
```

```
                        sprintf(str, "Input parameter '-w' should be an integer larger than 0!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;
                case 'S':
                    (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
                    break;
                case 'C':
                    (*param).CLONE_PROBABILITY = value[i];
                    if ((*param).CLONE_PROBABILITY < 0)
                        (*param).CLONE_PROBABILITY = 0;
                    else if ((*param).CLONE_PROBABILITY > 1)
                        (*param).CLONE_PROBABILITY = 1;
                    break;
                case 'y':
                    (*param).CLONE_PROBABILITY_DECAY = value[i];
                    if ((*param).CLONE_PROBABILITY_DECAY < 0)
                        (*param).CLONE_PROBABILITY_DECAY = 0;
                    else if ((*param).CLONE_PROBABILITY_DECAY > 1)
                        (*param).CLONE_PROBABILITY_DECAY = 1;
                    break;

                default:
                    sprintf(str, "Error, unknown input parameter '-%c'!\nRun 'budgetedsvm_train()' for help.", option[i]);
                    mexErrMsgTxt(str);
                    break;
            }
        }

        // for BSGD, when we use merging budget maintenance strategy then only Gaussian kernel can be used,
        //  due to the nature of merging; here check if user specified some other kernel while merging
        if (((*param).ALGORITHM == BSGD) && ((*param).KERNEL != KERNEL_FUNC_GAUSSIAN) && ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE))
        {
            mexPrintf("Warning, BSGD with merging strategy can only use Gaussian kernel!\nKernel function switched to Gaussian.\n");
            (*param).KERNEL = KERNEL_FUNC_GAUSSIAN;
        }

        // check the MAINTENANCE_SAMPLING_STRATEGY validity
        if ((*param).ALGORITHM == LLSVM)
        {
            if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 2)
            {
                // 0 - random removal, 1 - k-means, 2 - k-medoids
                sprintf(str, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm_train()' for help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
                mexErrMsgTxt(str);
            }
        }
        else if ((*param).ALGORITHM == BSGD)
        {
            if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 1)
            {
                // 0 - smallest removal, 1 - merging
                sprintf(str, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm_train()' for help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
                mexErrMsgTxt(str);
            }
        }

        // no bias term for LLSVM and BSGD functions
        if (((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD))
            (*param).BIAS_TERM = 0.0;

        if ((*param).VERBOSE)
        {
            mexPrintf("*** Training started with the following parameters:\n");
            switch ((*param).ALGORITHM)
            {
                case PEGASOS:
                    mexPrintf("Algorithm \t\t\t\t: Pegasos\n");
                    break;
                case AMM_ONLINE:
                    mexPrintf("Algorithm \t\t\t\t: AMM online\n");
                    break;
                case AMM_BATCH:
                    mexPrintf("Algorithm \t\t\t\t: AMM batch\n");
                    break;
                case BSGD:
                    mexPrintf("Algorithm \t\t\t\t\t: BSGD\n");
                    break;
                case LLSVM:
                    mexPrintf("Algorithm \t\t\t\t\t: LLSVM\n");
```

```
                break;
        }

        if (((*param).ALGORITHM == PEGASOS) || ((*param).ALGORITHM == AMM_BATCH) || ((*param).ALGORITHM == AMM_ONLINE))
        {
            mexPrintf("Lambda parameter \t\t: %f\n", (*param).LAMBDA_PARAM);
            mexPrintf("Bias term \t\t\t: %f\n", (*param).BIAS_TERM);
            if ((*param).ALGORITHM != PEGASOS)
            {
                mexPrintf("Pruning frequency k \t: %d\n", (*param).K_PARAM);
                mexPrintf("Pruning threshold c \t: %f\n", (*param).C_PARAM);
                mexPrintf("Num. weights per class\t: %d\n", (*param).BUDGET_SIZE);
                mexPrintf("Number of epochs \t\t: %d\n\n", (*param).NUM_EPOCHS);
            }
            else
                mexPrintf("\n");
        }
        else if (((*param).ALGORITHM == BSGD) || ((*param).ALGORITHM == LLSVM))
        {
            if ((*param).ALGORITHM == BSGD)
            {
                mexPrintf("Number of epochs \t\t\t: %d\n", (*param).NUM_EPOCHS);
                mexPrintf("Size of the budget \t\t\t: %d\n", (*param).BUDGET_SIZE);
                if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_REMOVE)
                    mexPrintf("Maintenance strategy \t\t: smallest removal)n");
                else if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE)
                    mexPrintf("Maintenance strategy \t\t: merging\n");
                else
                    mexErrMsgTxt("Error, unknown budget maintenance set. Run 'budgetedsvm_train()' for help.\n");

                mexPrintf("Lambda regularization param.: %f\n", (*param).LAMBDA_PARAM);
            }
            else if ((*param).ALGORITHM == LLSVM)
            {
                switch ((*param).MAINTENANCE_SAMPLING_STRATEGY)
                {
                    case LANDMARK_SAMPLE_RANDOM:
                        mexPrintf("Landmark sampling \t\t\t: random sampling\n");
                        break;

                    case LANDMARK_SAMPLE_KMEANS:
                        mexPrintf("Landmark sampling \t\t\t: k-means initialization\n");
                        break;

                    case LANDMARK_SAMPLE_KMEDOIDS:
                        mexPrintf("Landmark sampling \t\t\t: k-medoids initialization\n");
                        break;

                    default:
                        mexErrMsgTxt("Error, unknown landmark sampling set. Run 'budgetedsvm_train()' for help.\n");
                        break;
                }
                mexPrintf("Number of landmark points \t: %d\n", (*param).BUDGET_SIZE);
                mexPrintf("Lambda regularization param.: %f\n", (*param).LAMBDA_PARAM);
            }

            // print common parameters
            switch ((*param).KERNEL)
            {
                case KERNEL_FUNC_GAUSSIAN:
                    mexPrintf("Gaussian kernel used \t\t: K(x, y) = exp(-0.5 * gamma * ||x - y||^2)\n");
                    if ((*param).KERNEL_GAMMA_PARAM != 0.0)
                    {
                        sprintf(str, "Gaussian kernel width \t\t: %f\n\n", (*param).KERNEL_GAMMA_PARAM);
                        mexPrintf(str);
                    }
                    else
                        mexPrintf("Gaussian kernel width \t\t: 1 / DIMENSIONALITY\n\n");
                    break;

                case KERNEL_FUNC_EXPONENTIAL:
                    mexPrintf("Exponential kernel used \t: K(x, y) = exp(-0.5 * gamma * ||x - y||)\n");
                    if ((*param).KERNEL_GAMMA_PARAM != 0.0)
                    {
                        sprintf(str, "Exponential kernel width \t: %f\n\n", (*param).KERNEL_GAMMA_PARAM);
                        mexPrintf(str);
                    }
                    else
                        mexPrintf("Exponential kernel width \t: 1 / DIMENSIONALITY\n\n");
                    break;
```

```

                    case KERNEL_FUNC_POLYNOMIAL:
                        sprintf(str, "Polynomial kernel used \t\t: K(x, y) = (x^T * y + %.2f)^%.2f\n\n", (*param).KERNEL_COEF_PARAM, (*param).KERNEL_DEGREE_PARAM);
                        mexPrintf(str);
                        break;

                    case KERNEL_FUNC_SIGMOID:
                        sprintf(str, "Sigmoid kernel used \t\t: K(x, y) = tanh(%.2f * x^T * y + %.2f)\n\n", (*param).KERNEL_DEGREE_PARAM, (*param).KERNEL_COEF_PARAM);
                        mexPrintf(str);
                        break;

                    case KERNEL_FUNC_LINEAR:
                        mexPrintf("Linear kernel used \t\t\t: K(x, y) = (x^T * y)\n\n");
                        break;

                    case KERNEL_FUNC_USER_DEFINED:
                        mexPrintf("User-defined kernel function used.\n\n");
                        break;

                    default:
                        sprintf(str, "Input parameter '-K %d' out of bounds!\nRun 'budgetedsvm_train()' for help.\n", (*param).KERNEL);
                        mexErrMsgTxt(str);
                        break;
                }
            }
            mexEvalString("drawnow;");
        }

        // if inputs to training phase are .txt files, then also increase dimensionality due to added bias term, and update KERNEL_GAMMA_PARAM if not set by a user;
        //  NOTE that we do not execute this part if inputs are Matlab variables, as we still do not know the dimensionality, therefore BIAS_TERM and
        //  KERNEL_GAMMA_PARAM are adjusted in budgetedDataMatlab::readDataFromMatlab() function, after we find out the dimensionality of the considered data set
        if (inputFileName)
        {
            // signal error if a user wants to use an RBF kernel, but didn't specify either data dimension or kernel width
            if ((((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD)) && (((*param).KERNEL == KERNEL_FUNC_GAUSSIAN) || ((*param).KERNEL == KERNEL_FUNC_EXPONENTIAL)))
            {
                if (((*param).KERNEL_GAMMA_PARAM == 0.0) && ((*param).DIMENSION == 0))
                {
                    // this means that both RBF kernel width and dimension were not set by the user in the input string to the toolbox
                    //  since in this case the default value of RBF kernel is 1/dimensionality, report error to the user
                    mexErrMsgTxt("Error, RBF kernel in use, please set either kernel width or dimensionality!\nRun 'budgetedsvm_train()' for help.\n");
                }
            }

            // increase dimensionality if bias term included
            if ((*param).BIAS_TERM != 0.0)
            {
                (*param).DIMENSION++;
            }

            // set gamma to default value of dimensionality
            if ((*param).KERNEL_GAMMA_PARAM == 0.0)
                (*param).KERNEL_GAMMA_PARAM = 1.0 / (double) (*param).DIMENSION;
        }
    }
    else
    {
        // check if the model file exists only if model filename is provided
        if (modelFileName)
        {
            if (!readableFileExists(modelFileName))
            {
                sprintf(str, "Can't open model file %s!\n", modelFileName);
                mexErrMsgTxt(str);
            }
        }

        // modify parameters
        for (unsigned int i = 0; i < option.size(); i++)
        {
            switch (option[i])
            {
                /*case 'p':
                    (*param).SAVE_PREDS = (value[i] != 0);
                    break;*/
                case 'v':
```

```
                    (*param).VERBOSE = (value[i] != 0);
                    break;

                case 'z':
                    (*param).CHUNK_SIZE = (unsigned int) value[i];
                    if ((*param).CHUNK_SIZE < 1)
                    {
                        sprintf(str, "Input parameter '-z' should be an integer larger than 0!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;
                case 'w':
                    (*param).CHUNK_WEIGHT = (unsigned int) value[i];
                    if ((*param).CHUNK_WEIGHT < 1)
                    {
                        sprintf(str, "Input parameter '-w' should be an integer larger than 0!\nRun 'budgetedsvm_train()' for help.");
                        mexErrMsgTxt(str);
                    }
                    break;
                case 'S':
                    (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
                    break;

                default:
                    sprintf(str, "Error, unknown input parameter '-%c'!\nRun 'budgetedsvm_predict()' for help.", option[i]);
                    mexErrMsgTxt(str);
                    break;
            }
        }

        /*if ((*param).VERBOSE)
        {
            mexPrintf("\n*** Testing with the following parameters:\n");
            switch ((*param).ALGORITHM)
            {
                case PEGASOS:
                    mexPrintf("Algorithm: \t\t\t\tPEGASOS\n");
                    break;
                case AMM_ONLINE:
                    mexPrintf("Algorithm: \t\t\t\tAMM online\n");
                    break;
                case AMM_BATCH:
                    mexPrintf("Algorithm: \t\t\t\tAMM batch\n");
                    break;
                case BSGD:
                    mexPrintf("Algorithm: \t\t\t\tBSGD\n");
                    break;
            }

            if (((*param).ALGORITHM == PEGASOS) || ((*param).ALGORITHM == AMM_BATCH) || ((*param).ALGORITHM == AMM_ONLINE))
            {
                mexPrintf("Bias term: \t\t\t\t%f\n\n", (*param).BIAS_TERM);
            }
            else if ((*param).ALGORITHM == BSGD)
            {
                mexPrintf("Gaussian kernel width: \t%f\n\n", (*param).GAMMA_PARAM);
            }
            mexEvalString("drawnow;");
        }*/
    }

    setPrintErrorStringFunction(&printErrorStringMatlab);
    if ((*param).VERBOSE)
        setPrintStringFunction(&printStringMatlab);
    else
        setPrintStringFunction(NULL);
}
```

6.1.2.3 printErrorStringMatlab()

```
void printErrorStringMatlab (
        const char * s )
```

Prints error string to Matlab, used to modify callback found in budgetedSVM.cpp.

Parameters

in	*s*	Text to be printed.

Definition at line 1109 of file budgetedSVM_matlab.cpp.

```
{
    mexErrMsgTxt(s);
}
```

6.1.2.4 printStringMatlab()

```
void printStringMatlab (
            const char * s )
```

Prints string to Matlab, used to modify callback in budgetedSVM.cpp.

Parameters

in	*s*	Text to be printed.

Definition at line 1099 of file budgetedSVM_matlab.cpp.

```
{
    mexPrintf(s);
    mexEvalString("drawnow;");
}
```

6.1.2.5 printUsageMatlab()

```
void printUsageMatlab (
            bool trainingPhase,
            parameters * param )
```

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

Parameters

in	*trainingPhase*	Indicator if training or testing phase.
in	*param*	Parameter object modified by user input.

Definition at line 1127 of file budgetedSVM_matlab.cpp.

```
{
    if (trainingPhase)
    {
        mexPrintf("\n\tUsage:\n");
        mexPrintf("\t\tmodel = budgetedsvm_train(label_vector, instance_matrix, parameter_string = ")\n\n");
        mexPrintf("\tInputs:\n");
        mexPrintf("\t\tlabel_vector\t\t- label vector of size (NUM_POINTS x 1), a label set can include any integer\n");
        mexPrintf("\t\t\t\t\t               representing a class, such as 0/1 or +1/-1 in the case of binary-class\n");
        mexPrintf("\t\t\t\t\t               problems; in the case of multi-class problems it can be any set of integers\n");
```

```
        mexPrintf("\t\tinstance_matrix\t\t- instance matrix of size (NUM_POINTS x DIMENSIONALITY),\n");
        mexPrintf("\t\t\t\t                  where each row represents one example\n");
        mexPrintf("\t\tparameter_string\t- parameters of the model, defaults to empty string if not provided\n\n");
        mexPrintf("\tOutput:\n");
        mexPrintf("\t\tmodel\t\t\t\t- structure that holds the learned model\n\n");
        mexPrintf("\t--------------------------------------------\n\n");
        mexPrintf("\tIf the data set cannot be fully loaded to Matlab, another variant can be used:\n");
        mexPrintf("\t\tbudgetedsvm_train(train_file, model_file, parameter_string = ”)\n\n");
        mexPrintf("\tInputs:\n");
        mexPrintf("\t\ttrain_file\t\t\t- filename of .txt file containing training data set in LIBSVM format\n");
        mexPrintf("\t\tmodel_file\t\t\t- filename of .txt file that will contain trained model\n");
        mexPrintf("\t\tparameter_string\t- parameters of the model, defaults to empty string if not provided\n\n");
        mexPrintf("\t--------------------------------------------\n\n");
        mexPrintf("\tParameter string is of the following format:\n");
        mexPrintf("\t’-OPTION1 VALUE1 -OPTION2 VALUE2 ...’\n\n");
        mexPrintf("\tFollowing options are available; affected algorithm and default values\n");
        mexPrintf("\tare given in parentheses (algorithm not specified if option affects all):\n");
        mexPrintf("\t A - algorithm, which large-scale SVM to use (%d):\n", (*param).ALGORITHM);
        mexPrintf("\t\t     0 - Pegasos\n");
        mexPrintf("\t\t     1 - AMM batch\n");
        mexPrintf("\t\t     2 - AMM online\n");
        mexPrintf("\t\t     3 - LLSVM\n");
        mexPrintf("\t\t     4 - BSGD\n");
        mexPrintf("\t D - dimensionality (faster loading if set, if omitted inferred from the data)\n");
        mexPrintf("\t B - limit on the number of weights per class in AMM, OR\n");
        mexPrintf("\t\t     total SV set budget in BSGD, OR number of landmark points in LLSVM (%d)\n", (*param).BUDGET_SIZE);
        mexPrintf("\t L - lambda regularization parameter; high value -> less complex model (%.5f)\n", (*param).LAMBDA_PARAM);
        mexPrintf("\t b - bias term, if 0 no bias added (%.1f)\n", (*param).BIAS_TERM);
        mexPrintf("\t e - number of training epochs (AMM, BSGD; %d)\n", (*param).NUM_EPOCHS);
        mexPrintf("\t s - number of subepochs (AMM batch; %d)\n", (*param).NUM_SUBEPOCHS);
        mexPrintf("\t k - pruning frequency, after how many observed examples is pruning done (AMM; %d)\n", (*param).K_PARAM);
        mexPrintf("\t c - pruning threshold; high value -> less complex model (AMM; %.2f)\n", (*param).C_PARAM);
        mexPrintf("\t K - kernel function (0 - RBF; 1 - exponential, 2 - polynomial; 3 - linear, \n");
        mexPrintf("\t\t     4 - sigmoid; 5 - user-defined) (LLSVM, BSGD; %d)\n", (*param).KERNEL);
        mexPrintf("\t g - RBF or exponential kernel width gamma (LLSVM, BSGD; 1/DIMENSIONALITY)\n");
        mexPrintf("\t d - polynomial kernel degree or sigmoid kernel slope (LLSVM, BSGD; %.2f)\n", (*param).KERNEL_DEGREE_PARAM);
        mexPrintf("\t i - polynomial or sigmoid kernel intercept (LLSVM, BSGD; %.2f)\n", (*param).KERNEL_COEF_PARAM);
        mexPrintf("\t m - budget maintenance in BSGD (0 - removal; 1 - merging, uses Gaussian kernel), OR\n");
        mexPrintf("\t\t     landmark sampling strategy in LLSVM (0 - random; 1 - k-means; 2 - k-medoids) (%d)\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
        mexPrintf("\t C - clone probability when misclassification occurs in AMM (%d)\n", (*param).CLONE_PROBABILITY);
        mexPrintf("\t y - clone probability decay when weight cloning occurs in AMM (%.2f)\n\n", (*param).CLONE_PROBABILITY_DECAY);

        mexPrintf("\t z - training and test file are loaded in chunks so that the algorithm can \n");
        mexPrintf("\t\t     handle budget files on weaker computers; z specifies number of examples loaded in\n");
        mexPrintf("\t\t     a single chunk of data, ONLY when inputs are .txt files (%d)\n", (*param).CHUNK_SIZE);
        mexPrintf("\t w - model weights are split in chunks, so that the algorithm can handle\n");
        mexPrintf("\t\t     highly dimensional data on weaker computers; w specifies number of dimensions stored\n");
        mexPrintf("\t\t     in one chunk, ONLY when inputs are .txt files (%d)\n", (*param).CHUNK_WEIGHT);
        mexPrintf("\t S - if set to 1 data is assumed sparse, if 0 data is assumed non-sparse, used to\n");
        mexPrintf("\t\t     speed up kernel computations (default is 1 when percentage of non-zero\n");
        mexPrintf("\t\t     features is less than 5%%, and 0 when percentage is larger than 5%%)\n");
        mexPrintf("\t r - randomize the algorithms; 1 to randomize, 0 not to randomize (%d)\n", (*param).RANDOMIZE);
        mexPrintf("\t v - verbose output: 1 to show the algorithm steps (epoch ended, training started, ...), 0 for quiet mode (%d)\n", (*param).VERBOSE);
        mexPrintf("\t--------------------------------------------\n");
        mexPrintf("\tInstructions on how to convert data to and from the LIBSVM format can be found on <a href=\"http://www.csie.ntu.edu.tw/~cjlin/libsvm/\">LIBSVM website</a>.\n");
    }
    else
    {
        mexPrintf("\n\tUsage:\n");
        mexPrintf("\t\t[error_rate, pred_labels, pred_scores] = budgetedsvm_predict(label_vector, instance_matrix, model, parameter_string = ”)\n\n");
        mexPrintf("\tInputs:\n");
        mexPrintf("\t\tlabel_vector\t\t- label vector of size (NUM_POINTS x 1), a label set can include any integer\n");
```

```
        mexPrintf("\t\t\t\t\t             representing a class, such as 0/1 or +1/-1 in the case of binary-class\n");
        mexPrintf("\t\t\t\t\t             problems; in the case of multi-class problems it can be any set of integers\n");
        mexPrintf("\t\tinstance_matrix\t\t- instance matrix of size (NUM_POINTS x DIMENSIONALITY),\n");
        mexPrintf("\t\t\t\t                 where each row represents one example\n");
        mexPrintf("\t\tmodel\t\t\t\t- structure holding the model learned through budgetedsvm_train()\n");
        mexPrintf("\t\tparameter_string\t- parameters of the model, defaults to empty string if not provided\n\n");
        mexPrintf("\tOutput:\n");
        mexPrintf("\t\terror_rate\t\t\t- error rate on the test set\n");
        mexPrintf("\t\tpred_labels\t\t\t- vector of predicted labels of size (NUM_POINTS x 1)\n");
        mexPrintf("\t\tpred_scores\t\t\t- vector of predicted scores of size (NUM_POINTS x 1)\n\n");

        mexPrintf("\t--------------------------------------------\n\n");

        mexPrintf("\tIf the data set cannot be fully loaded to Matlab, another variant can be used:\n");
        mexPrintf("\t\t[error_rate, pred_labels, pred_scores] = budgetedsvm_predict(test_file, model_file, parameter_string = ”)\n\n");
        mexPrintf("\tInputs:\n");
        mexPrintf("\t\ttest_file\t\t\t- filename of .txt file containing test data set in LIBSVM format\n");
        mexPrintf("\t\tmodel_file\t\t\t- filename of .txt file containing model trained through budgetedsvm_train()\n");
        mexPrintf("\t\tparameter_string\t- parameters of the model, defaults to empty string if not provided\n\n");
        mexPrintf("\tOutput:\n");
        mexPrintf("\t\terror_rate\t\t\t- error rate on the test set\n");
        mexPrintf("\t\tpred_labels\t\t\t- vector of predicted labels of size (NUM_POINTS x 1)\n");
        mexPrintf("\t\tpred_scores\t\t\t- vector of predicted scores of size (NUM_POINTS x 1)\n\n");

        mexPrintf("\t--------------------------------------------\n\n");
        mexPrintf("\tParameter string is of the following format:\n");
        mexPrintf("\t’-OPTION1 VALUE1 -OPTION2 VALUE2 ...’\n\n");
        mexPrintf("\tThe following options are available (default values in parentheses):\n");
        mexPrintf("\tz - the training and test file are loaded in chunks so that the algorithm can\n");
        mexPrintf("\t\t    handle budget files on weaker computers; z specifies number of examples loaded in\n");
        mexPrintf("\t\t    a single chunk of data, ONLY when inputs are .txt files (%d)\n", (*param).CHUNK_SIZE);
        mexPrintf("\tw - the model weight is split in parts, so that the algorithm can handle\n");
        mexPrintf("\t\t    highly dimensional data on weaker computers; w specifies number of dimensions stored\n");
        mexPrintf("\t\t    in one chunk, ONLY when inputs are .txt files (%d)\n", (*param).CHUNK_WEIGHT);
        mexPrintf("\tS - if set to 1 data is assumed sparse, if 0 data is assumed non-sparse, used to\n");
        mexPrintf("\t\t    speed up kernel computations (default is 1 when percentage of non-zero\n");
        mexPrintf("\t\t    features is less than 5%%, and 0 when percentage is larger than 5%%)\n");
        mexPrintf("\tv - verbose output: 1 to show algorithm steps, 0 for quiet mode (%d)\n", (*param).VERBOSE);
        mexPrintf("\t--------------------------------------------\n");
        mexPrintf("\tInstructions on how to convert data to and from the LIBSVM format can be found on <a href=\"http://www.csie.ntu.edu.tw/~cjlin/libsvm/\">LIBSVM website</a>.\n");
    }
}
```

6.2 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/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	*testData*	Input test data.
in	*param*	The parameters of the algorithm.
in	*model*	Trained BSGD model.
out	*labels*	Vector of predicted labels.
out	*scores*	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, outputing the predicted labels and the error rate.

Definition at line 513 of file bsgd.cpp.

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

```
    budgetedVectorBSGD *currentDataPoint = NULL;
    long double *classMaxScores = new long double[(testData->yLabels).size()];

    // this tempArray is used when calculating all class scores, to avoid repeated computations of the same kernel
    tempArray = new long double[(*(model->modelBSGD)).size()];
    for (i = 0; i < (*(model->modelBSGD)).size(); i++)
        tempArray[i] = 0.0;

    while (stillChunksLeft)
    {
        stillChunksLeft = testData->readChunk((*param).CHUNK_SIZE);
        (*param).updateVerySparseDataParameter(testData->getSparsity());

        N = testData->N;
        total += N;
        start = clock();

        for (unsigned int r = 0; r < N; r++)
        {
            if ((*param).VERY_SPARSE_DATA)
            {
                // since we are computing kernels using vectors directly from the budgetedData, we need square norm of the vector to speed-up
                //  computations, here we compute it just once; no need to do it in non-sparse case, since this norm can be retrieved directly
                //  from budgetedVector
                tempSqrNorm = testData->getVectorSqrL2Norm(r, param);
            }
            else
            {
                // create the budgetedVector using the vector from budgetedData, to be used in kernel computations below
                currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT, (unsigned int) (testData->yLabels).size());
                currentDataPoint->createVectorUsingDataPoint(testData, r, param);
            }

            y = 0;
            maxFx = -INF;
            for (i = 0; i < (*(model->modelBSGD)).size(); i++)
                tempArray[i] = 0.0;

            for (unsigned int k = 0; k < (testData->yLabels).size(); k++)
            {
                classMaxScores[k] = -INF;
                fx = 0;
                for (unsigned int i = 0; i < (*(model->modelBSGD)).size(); i++)
                {
                    if ((*(model->modelBSGD))[i]->alphas[k] != 0)
                    {
                        if (tempArray[i] == 0.0)
                        {
                            if ((*param).VERY_SPARSE_DATA)
                            {
                                // directly compute kernel from the trainData
                                tempArray[i] = (*((*model).modelBSGD))[i]->computeKernel(r, testData, param, tempSqrNorm);
                            }
                            else
                            {
                                // compute kernel from currentDataPoint object
                                tempArray[i] = (*(model->modelBSGD))[i]->computeKernel(currentDataPoint, param);
                            }
                        }
                        fx += ((*(model->modelBSGD))[i]->alphas[k] * tempArray[i]);
                    }
                }

                if (fx > maxFx)
                {
                    maxFx = fx;
                    y = k;
                }

                if (fx > classMaxScores[k])
                    classMaxScores[k] = fx;
            }

            if (!(*param).VERY_SPARSE_DATA)
            {
                // if sparse then no need for this, since we didn't even create currentDataPoint
                delete currentDataPoint;
                currentDataPoint = NULL;
            }

```

```
            if (y != testData->al[r])
                err++;

            // save predicted label, will be sent to output
            if (labels)
                (*labels).push_back((int) (testData->yLabels)[y]);
            // ... and the scores
            if (scores)
            {
                // for BSGD the output score is the difference between winning and the second best score
                long double secondBestScore = -INF;
                for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
                {
                    if (i == y)
                        continue;

                    if (secondBestScore < classMaxScores[i])
                        secondBestScore = classMaxScores[i];
                }
                (*scores).push_back((float) (maxFx - secondBestScore));
            }
        }

        timeCalc += clock() - start;

        if (((*param).VERBOSE) && (N > 0))
        {
            sprintf(text, "Number of examples processed: %d\n", total);
            svmPrintString(text);
        }
    }
    testData->flushData();
    delete [] tempArray;
    delete[] classMaxScores;

    if ((*param).VERBOSE)
    {
        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);
        svmPrintString(text);
    }

    return (float) (100.0 * (float)err / (float)total);
}
```

6.2.2.2 trainBSGD()

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

Train BSGD.

Parameters

in	*trainData*	Input training data.
in	*param*	The parameters of the algorithm.
in,out	*model*	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.

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

```
    unsigned int i1, i2 = 0, t, temp, countDel = 0, numClasses = 0, numSVs = 0, numIter = 0, N, deleteWeight = 0;
    bool stillChunksLeft = true;
    char text[1024];
    unsigned int i, k, ot;  //iterators
    budgetedVectorBSGD *currentDataPoint = NULL;
    int indexOfSameVector = -1; // this variable keeps the index of the *exact same* vector in the SV set, when compared to input point.
                                //  so when we observe budget overflow we merge these two if merging strategy is set

    // this tempArray is used when calculating all class scores and runner-up, to avoid repeated computations of the same kernel
    tempArray = new long double[(*param).BUDGET_SIZE];
    for (i = 0; i < (*param).BUDGET_SIZE; i++)
        tempArray[i] = 0.0;

    for (unsigned int epoch = 0; epoch < (*param).NUM_EPOCHS; epoch++)
    {
        //Calculate
        stillChunksLeft = true;
        while (stillChunksLeft)
        {
            stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);

            // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
            //  (of course, in the case of AMM, speeds up linear kernel computation)
            (*param).updateVerySparseDataParameter(trainData->getSparsity());

            // compute observed data dimensionality, where we also account for possible bias term, and check if
            //  we need to expand the current model weights if some new data dimensions were found during loading
            temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
            if ((*param).DIMENSION < temp)
            {
                /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION, temp);
                svmPrintString(text);*/
                (*model).extendDimensionalityOfModel(temp, param);

                // update the dimensionality
                (*param).DIMENSION = temp;
            }

            N = trainData->N;
            if (numIter == 0)
                numClasses = (unsigned int) trainData->yLabels.size();
            else if (numClasses != (unsigned int) trainData->yLabels.size())
            {
                // if in the earlier chunks some class wasn't observed, it could happen with small chunks or unbalanced classes;
                // just add new zero alphas for the new classes to each support vector
                for (unsigned int i = 0; i < numSVs; i++)
                    for (unsigned int k = 0; k < (trainData->yLabels.size() - numClasses); k++)
                        (*((*model).modelBSGD))[i]->alphas.push_back(0.0);
                numClasses = (unsigned int) trainData->yLabels.size();
            }

            // randomize
            vector <unsigned int> tv(N, 0);
            for (unsigned int ti = 0; ti < N; ti++)
            {
                tv[ti] = ti;
            }
            if ((*param).RANDOMIZE)
                random_shuffle(tv.begin(), tv.end());

            start = clock();
            for (ot = 0; ot < N; ot++)
            {
                t = tv[ot];
                numIter++;

                // initialize the first weight
                if (numIter == 1)
                {
                    currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT, numClasses);
                    currentDataPoint->createVectorUsingDataPoint(trainData, t, param);

                    i1 = trainData->al[t];
                    i2 = (i1 + 1) % numClasses;
                    currentDataPoint->alphas[i1] = 1.0;
                    currentDataPoint->alphas[i2] = -1.0;

                    (*((*model).modelBSGD)).push_back(currentDataPoint);
```

```
                currentDataPoint = NULL;
                numSVs++;
                continue;
            }

            // calculate all class scores and runner-up
            i1 = trainData->al[t];
            fxValue1 = 0.0;
            fxValue2 = 0.0;
            maxFx = -INF;
            for (i = 0; i < numSVs; i++)
                tempArray[i] = 0.0;

            if ((*param).VERY_SPARSE_DATA)
            {
                // since we are computing kernels using vectors directly from the budgetedData, we need square norm of the vector to speed-up
                //  computations, here we compute it just once; no need to do it in non-sparse case, since this norm can be retrieved directly
                //  from budgetedVector
                tempLongDouble = trainData->getVectorSqrL2Norm(t, param);
            }
            else
            {
                // create the budgetedVector using the vector from budgetedData, to be used in gaussianKernel() method below
                currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT, numClasses);
                currentDataPoint->createVectorUsingDataPoint(trainData, t, param);
            }

            indexOfSameVector = -1;
            for (k = 0; k < numClasses; k++)
            {
                fxValue = 0.0;
                for (i = 0; i < numSVs; i++)
                {
                    if ((*((*model).modelBSGD))[i]->alphas[k] != 0)
                    {
                        // calculate the kernel only if not computed earlier
                        if (tempArray[i] == 0.0)
                        {
                            if ((*param).VERY_SPARSE_DATA)
                            {
                                // directly compute kernel from the trainData
                                tempArray[i] = (*((*model).modelBSGD))[i]->computeKernel(t, trainData, param, tempLongDouble);

                                // check if the two vectors are identical, if they are then we consider these two vectors when budget overflow
                                //  happens (we round to 8th digit due to round-off errors in floating-point representation, observed in practice)
                                if ((int) ((*((*model).modelBSGD))[i]->gaussianKernel(t, trainData, param, tempLongDouble) * 100000000.0 + 0.5) == 100000000)
                                {
                                    indexOfSameVector = i;
                                }
                            }
                            else
                            {
                                // compute kernel from currentDataPoint object
                                tempArray[i] = (*((*model).modelBSGD))[i]->computeKernel(currentDataPoint, param);

                                // check if the two vectors are identical, if they are then we consider these two vectors when budget overflow
                                //  happens (we round to 8th digit due to round-off errors in floating-point representation, observed in practice)
                                if ((int) ((*((*model).modelBSGD))[i]->gaussianKernel(currentDataPoint, param) * 100000000.0 + 0.5) == 100000000)
                                {
                                    indexOfSameVector = i;
                                }
                            }
                        }
                        fxValue += ((*((*model).modelBSGD))[i]->alphas[k] * tempArray[i]);
                    }
                }

                if (k == i1)
                    fxValue1 = fxValue;
                else if (fxValue > maxFx)
                {
                    maxFx = fxValue;
                    i2 = k;
                }
```

```
            }
            fxValue2 = maxFx;

            // downweight all the weights
            for (i = 0; i < numSVs; i++)
                (*((*model).modelBSGD))[i]->downgrade(numIter);

            if (1.0 + fxValue2 - fxValue1 > 0.0)
            {
                if ((*param).VERY_SPARSE_DATA)
                {
                    // only do this if data is sparse, since if non-sparse than we already have currentDataPoint initialized
                    //  from the code before the loop in which we computed kernels
                    currentDataPoint = new budgetedVectorBSGD((*param).DIMENSION, (*param).CHUNK_WEIGHT, numClasses);
                    currentDataPoint->createVectorUsingDataPoint(trainData, t, param);
                }

                // add an SV
                currentDataPoint->alphas[i1] =  1.0 / ((long double)numIter * (*param).LAMBDA_PARAM);
                currentDataPoint->alphas[i2] = -1.0 / ((long double)numIter * (*param).LAMBDA_PARAM);
                (*((*model).modelBSGD)).push_back(currentDataPoint);
                currentDataPoint = NULL;
                numSVs++;

                // if over the budget, maintain the budget
                if (numSVs > (*param).BUDGET_SIZE)
                {
                    switch ((*param).MAINTENANCE_SAMPLING_STRATEGY)
                    {
                        case BUDGET_MAINTAIN_REMOVE:
                            // removal of random support vector
                            if (indexOfSameVector == -1)
                            {
                                // so there are no two identical vectors, remove the smallest one
                                alphaSmallest = INF;
                                for (i = 0; i < numSVs; i++)
                                {
                                    // compute product between norm of alpha vector and a self-kernel
                                    tempLongDouble = (*((*model).modelBSGD))[i]->alphaNorm() * (*((*model).modelBSGD))[i]->computeKernel((*((*model).modelBSGD))[i], param);
                                    if (alphaSmallest > tempLongDouble)
                                    {
                                        alphaSmallest = tempLongDouble;
                                        deleteWeight = i;
                                    }
                                }
                            }
                            else
                            {
                                // since there is already an identical vector in the SV set, remove the newly added vector
                                deleteWeight = (*param).BUDGET_SIZE;
                            }

                            delete (*((*model).modelBSGD))[deleteWeight];
                            (*((*model).modelBSGD)).erase((*((*model).modelBSGD)).begin() + deleteWeight);
                            break;

                        case BUDGET_MAINTAIN_MERGE:
                            // merging of two SVs
                            long double kMax, kZ1, kZ2;
                            unsigned int merge1, merge2;

                            if (indexOfSameVector == -1)
                            {
                                // so there are no two identical vectors. find the one with smallest alpha
                                // here we look for who to merge
                                merge1 = 0;
                                for (i = 0; i < numSVs; i++)
                                {
                                    tempLongDouble = (*((*model).modelBSGD))[i]->alphaNorm();
                                    if ((alphaSmallest > tempLongDouble) || (i == 0))
                                    {
                                        alphaSmallest = tempLongDouble;
                                        merge1 = i;
                                    }
                                }

                                // find with who to merge, as well as other useful information detailed in the definition of computeKmax() found in this file
```

```
                                    long double* returnValues = computeKmax((*model).modelBSGD, merge1, param);
                                    kMax = (*returnValues);
                                    kZ1 = (*(returnValues + 1));
                                    kZ2 = (*(returnValues + 2));
                                    merge2 = (unsigned int) (*(returnValues + 3));
                                    delete [] returnValues;

                                    // find z, the new support vector
                                    (*((*model).modelBSGD))[merge1]->updateSV((*((*model).modelBSGD))[merge2], kMax);
                                    for (unsigned int k = 0; k < numClasses; k++)
                                        (*((*model).modelBSGD))[merge1]->alphas[k] = (*((*model).modelBSGD))[merge1]->alphas[k] * kZ1 + (*((*model).modelBSGD))[merge2]->alphas[k] * kZ2;
                                }
                                else
                                {
                                    // in this case the two merging vectors are identical, so we simply add up their alphas and there is no need for moving the vector
                                    merge1 = indexOfSameVector;
                                    merge2 = (*param).BUDGET_SIZE;
                                    for (unsigned int k = 0; k < numClasses; k++)
                                        (*((*model).modelBSGD))[merge1]->alphas[k] = (*((*model).modelBSGD))[merge1]->alphas[k] + (*((*model).modelBSGD))[merge2]->alphas[k];
                                }

                                // delete 'merge2', not needed anymore
                                delete (*((*model).modelBSGD))[merge2];
                                (*((*model).modelBSGD)).erase((*((*model).modelBSGD)).begin() + merge2);
                                break;
                        }
                        numSVs--;
                        countDel++;
                    }
                }
                else
                {
                    if (!(*param).VERY_SPARSE_DATA)
                    {
                        // if sparse data then no need for this part, since we didn't even create currentDataPoint
                        delete currentDataPoint;
                        currentDataPoint = NULL;
                    }
                }
            }
            timeCalc += clock() - start;

            if (((*param).VERBOSE) && (N > 0))
            {
                sprintf(text, "Number of examples processed: %d\n", numIter);
                svmPrintString(text);
            }
        }

        if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
        {
            sprintf(text, "Epoch %d/%d done.\n", epoch + 1, (*param).NUM_EPOCHS);
            svmPrintString(text);
        }
    }
    delete [] tempArray;
    trainData->flushData();

    if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
        svmPrintString("\n");

    if ((*param).VERBOSE)
    {
        sprintf(text, "Training completed in %5.3f seconds.\n\nNumber of budget maintenance steps: %d\n", (double)timeCalc / (double)CLOCKS_PER_SEC, countDel);
        svmPrintString(text);
    }
}
```

6.3 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSVM/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**, **KER↩ NEL_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	*fHandle*	Handle to an open file from which one word is read.
out	*str*	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.

```
{
    char temp;
    unsigned char index = 0;
    bool wordStarted = false;
    while (1)//for (int i = 0; i < 20; i++)
    {
        temp = (char) fgetc(fHandle);
```

```

        if (temp == EOF)
        {
            str[index++] = '\0';
            return true;
        }

        switch (temp)
        {
            case ' ':
                if (wordStarted)
                {
                    str[index++] = '\0';
                    return false;
                }
                break;

            case '\n':
                str[index++] = '\0';
                return true;
                break;

            default:
                wordStarted = true;
                str[index++] = temp;
                break;
        }
    }
}
```

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

Definition at line 1303 of file budgetedSVM.cpp.

```
{
    vector <char> option;
    vector <float> value;
    int i;
    FILE *pFile = NULL;
    char text[1024];

    // parse options
    for (i = 1; i < argc; i++)
```

```
    {
        if (argv[i][0] != '-')
            break;
        ++i;
        option.push_back(argv[i - 1][1]);
        value.push_back((float) atof(argv[i]));
    }

    if (trainingPhase)
    {
        if (i >= argc)
        {
            svmPrintErrorString("Error, input format not recognized. Run 'budgetedsvm-train' for help.\n");
        }

        pFile = fopen(argv[i], "r");
        if (pFile == NULL)
        {
            sprintf(text, "Can't open input file %s!\n", argv[i]);
            svmPrintErrorString(text);
        }
        else
        {
            fclose(pFile);
            strcpy(inputFile, argv[i]);
        }

        // take model file if provided by a user
        if (i < argc - 1)
            strcpy(modelFile, argv[i + 1]);
        else
        {
            char *p = strrchr(argv[i], '/');
            if (p == NULL)
                p = argv[i];
            else
                ++p;
            sprintf(modelFile, "%s.model", p);
        }

        // modify parameters
        for (unsigned int i = 0; i < option.size(); i++)
        {
            switch (option[i])
            {
                case 'A':
                    (*param).ALGORITHM = (unsigned int) value[i];
                    if ((*param).ALGORITHM > 4)
                    {
                        sprintf(text, "Input parameter '-A %d' out of bounds!\nRun 'budgetedsvm-train' for help.\n", (*param).ALGORITHM);
                        svmPrintErrorString(text);
                    }
                    break;

                case 'e':
                    (*param).NUM_EPOCHS = (unsigned int) value[i];
                    break;

                case 'D':
                    (*param).DIMENSION = (unsigned int) value[i];
                    break;

                case 's':
                    (*param).NUM_SUBEPOCHS = (unsigned int) value[i];
                    break;

                case 'k':
                    (*param).K_PARAM = (unsigned int) value[i];
                    break;

                case 'c':
                    (*param).C_PARAM = (double) value[i];
                    if ((*param).C_PARAM < 0.0)
                    {
                        sprintf(text, "Input parameter '-c' should be a non-negative real number!\nRun 'budgetedsvm-train' for help.\n");
                        svmPrintErrorString(text);
                    }
                    break;

                case 'L':
                    (*param).LAMBDA_PARAM = (double) value[i];
                    if ((*param).LAMBDA_PARAM <= 0.0)
                    {
                        sprintf(text, "Input parameter '-L' should be a positive real number!\nRun
```

```
'budgetedsvm-train' for help.\n");
                svmPrintErrorString(text);
            }
            break;

        case 'B':
            (*param).BUDGET_SIZE = (unsigned int) value[i];
            if ((*param).BUDGET_SIZE < 1)
            {
                sprintf(text, "Input parameter '-B' should be a positive integer!\nRun 'budgetedsvm-train' for help.\n");
                svmPrintErrorString(text);
            }
            break;

        case 'g':
            (*param).KERNEL_GAMMA_PARAM = (double) value[i];
            if ((*param).KERNEL_GAMMA_PARAM <= 0.0)
            {
                sprintf(text, "Input parameter '-g' should be a positive real number!\nRun 'budgetedsvm-train' for help.\n");
                svmPrintErrorString(text);
            }
            break;

        case 'd':
            (*param).KERNEL_DEGREE_PARAM = (double) value[i];
            if ((*param).KERNEL_DEGREE_PARAM <= 0.0)
            {
                sprintf(text, "Input parameter '-d' should be a positive real number!\nRun 'budgetedsvm-train' for help.\n");
                svmPrintErrorString(text);
            }
            break;

        case 'i':
            (*param).KERNEL_COEF_PARAM = (double) value[i];
            break;

        case 'K':
            (*param).KERNEL = (unsigned int) value[i];
            if ((*param).KERNEL > 5)
            {
                sprintf(text, "Input parameter '-K %d' out of bounds!\nRun 'budgetedsvm-train' for help.\n", (*param).KERNEL);
                svmPrintErrorString(text);
            }
            break;

        case 'm':
            (*param).MAINTENANCE_SAMPLING_STRATEGY = (unsigned int) value[i];
            break;

        case 'b':
            (*param).BIAS_TERM = (double) value[i];
            break;

        case 'v':
            (*param).VERBOSE = (value[i] != 0);
            break;

        case 'z':
            (*param).CHUNK_SIZE = (unsigned int) value[i];
            if ((*param).CHUNK_SIZE < 1)
            {
                sprintf(text, "Input parameter '-z' should be a positive real number!\nRun 'budgetedsvm-train' for help.\n");
                svmPrintErrorString(text);
            }
            break;

        case 'w':
            (*param).CHUNK_WEIGHT = (unsigned int) value[i];
            if ((*param).CHUNK_WEIGHT < 1)
            {
                sprintf(text, "Input parameter '-w' should be a positive real number!\nRun 'budgetedsvm-train' for help.\n");
                svmPrintErrorString(text);
            }
            break;

        case 'S':
            (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
            break;

        case 'r':
            (*param).RANDOMIZE = (value[i] != 0);
```

```
                    break;

                case 'C':
                    (*param).CLONE_PROBABILITY = (double) value[i];
                    if (((*param).CLONE_PROBABILITY < 0.0) || ((*param).CLONE_PROBABILITY > 1.0))
                    {
                        sprintf(text, "Input parameter '-C' should be a real number between 0 and 1!\nRun 'budgetedsvm-train()' for help.\n");
                        svmPrintErrorString(text);
                    }
                    break;

                case 'y':
                    (*param).CLONE_PROBABILITY_DECAY = (double) value[i];
                    if (((*param).CLONE_PROBABILITY_DECAY < 0.0) || ((*param).CLONE_PROBABILITY_DECAY > 1.0))
                    {
                        sprintf(text, "Input parameter '-y' should be a real number between 0 and 1!\nRun 'budgetedsvm-train()' for help.\n");
                        svmPrintErrorString(text);
                    }
                    break;

                default:
                    sprintf(text, "Error, unknown input parameter '-%c'!\nRun 'budgetedsvm-train' for help.\n", option[i]);
                    svmPrintErrorString(text);
                    break;
            }
        }

        // for BSGD, when we use merging budget maintenance strategy then only Gaussian kernel can be used,
        //  due to the nature of merging; here check if user specified some other kernel while merging
        if (((*param).ALGORITHM == BSGD) && ((*param).KERNEL != KERNEL_FUNC_GAUSSIAN) && ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE))
        {
            svmPrintString("Warning, BSGD with merging strategy can only use Gaussian kernel!\nKernel function switched to Gaussian.\n");
            (*param).KERNEL = KERNEL_FUNC_GAUSSIAN;
        }

        // signal error if a user wants to use RBF kernel, but didn't specify either data dimension or kernel width
        if ((((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD)) && (((*param).KERNEL == KERNEL_FUNC_GAUSSIAN) || ((*param).KERNEL == KERNEL_FUNC_EXPONENTIAL)))
        {
            if (((*param).KERNEL_GAMMA_PARAM == 0.0) && ((*param).DIMENSION == 0))
            {
                // this means that both RBF kernel width and dimension were not set by the user in the input string to the toolbox
                //  since in this case the default value of RBF kernel is 1/dimensionality, report error to the user
                svmPrintErrorString("Error, RBF kernel in use, please set either kernel width or dimensionality!\nRun 'budgetedsvm-train' for help.\n");
            }
        }

        // check the MAINTENANCE_SAMPLING_STRATEGY validity
        if ((*param).ALGORITHM == LLSVM)
        {
            if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 2)
            {
                // 0 - random removal, 1 - k-means, 2 - k-medoids
                sprintf(text, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm-train' for help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
                svmPrintErrorString(text);
            }
        }
        else if ((*param).ALGORITHM == BSGD)
        {
            if ((*param).MAINTENANCE_SAMPLING_STRATEGY > 1)
            {
                // 0 - smallest removal, 1 - merging
                sprintf(text, "Error, unknown input parameter '-m %d'!\nRun 'budgetedsvm-train' for help.\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
                svmPrintErrorString(text);
            }
        }

        // shut down printing to screen if user specified so
        if (!(*param).VERBOSE)
            setPrintStringFunction(NULL);

        // no bias term for LLSVM and BSGD functions
        if (((*param).ALGORITHM == LLSVM) || ((*param).ALGORITHM == BSGD))
        {
```

```
            (*param).BIAS_TERM = 0.0;
        }

        if ((*param).VERBOSE)
        {
            svmPrintString("\n*** Training started with the following parameters:\n");
            switch ((*param).ALGORITHM)
            {
                case PEGASOS:
                    svmPrintString("Algorithm \t\t\t: Pegasos\n");
                    break;
                case AMM_ONLINE:
                    svmPrintString("Algorithm \t\t\t: AMM online\n");
                    break;
                case AMM_BATCH:
                    svmPrintString("Algorithm \t\t\t: AMM batch\n");
                    break;
                case BSGD:
                    svmPrintString("Algorithm \t\t\t: BSGD\n");
                    break;
                case LLSVM:
                    svmPrintString("Algorithm \t\t\t: LLSVM\n");
                    break;
            }

            if (((*param).ALGORITHM == PEGASOS) || ((*param).ALGORITHM == AMM_BATCH) || ((*param).ALGORITHM == AMM_ONLINE))
            {
                sprintf(text, "Lambda parameter\t\t: %f\n", (*param).LAMBDA_PARAM);
                svmPrintString(text);
                sprintf(text, "Bias term \t\t\t: %f\n", (*param).BIAS_TERM);
                svmPrintString(text);
                if ((*param).ALGORITHM != PEGASOS)
                {
                    sprintf(text, "Pruning frequency k \t\t: %d\n", (*param).K_PARAM);
                    svmPrintString(text);
                    sprintf(text, "Pruning parameter c \t\t: %.2f\n", (*param).C_PARAM);
                    svmPrintString(text);
                    sprintf(text, "Max num. of weights per class \t: %d\n", (*param).BUDGET_SIZE);
                    svmPrintString(text);
                    sprintf(text, "Number of epochs \t\t: %d\n\n", (*param).NUM_EPOCHS);
                    svmPrintString(text);
                }
                else
                    svmPrintString("\n");
            }
            else if (((*param).ALGORITHM == BSGD) || ((*param).ALGORITHM == LLSVM))
            {
                if ((*param).ALGORITHM == BSGD)
                {
                    sprintf(text, "Number of epochs \t\t: %d\n", (*param).NUM_EPOCHS);
                    svmPrintString(text);
                    if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_REMOVE)
                        svmPrintString("Maintenance strategy \t\t: 0 (smallest removal)\n");
                    else if ((*param).MAINTENANCE_SAMPLING_STRATEGY == BUDGET_MAINTAIN_MERGE)
                        svmPrintString("Maintenance strategy \t\t: 1 (merging)\n");
                    else
                        svmPrintErrorString("Error, unknown budget maintenance set. Run 'budgetedsvm-train' for help.\n");

                    svmPrintString(text);
                    sprintf(text, "Size of the budget \t\t: %d\n", (*param).BUDGET_SIZE);
                    svmPrintString(text);
                }
                else if ((*param).ALGORITHM == LLSVM)
                {
                    switch ((*param).MAINTENANCE_SAMPLING_STRATEGY)
                    {
                        case LANDMARK_SAMPLE_RANDOM:
                            svmPrintString("Landmark sampling \t\t: 0 (random sampling)\n");
                            break;
                        case LANDMARK_SAMPLE_KMEANS:
                            svmPrintString("Landmark sampling \t\t: k-means initialization\n");
                            break;
                        case LANDMARK_SAMPLE_KMEDOIDS:
                            svmPrintString("Landmark sampling \t\t: 1 (k-medoids initialization)\n");
                            break;
                        default:
                            svmPrintErrorString("Error, unknown landmark sampling set. Run 'budgetedsvm-train' for help.\n");
                            break;
                    }
                    sprintf(text, "Number of landmark points \t: %d\n", (*param).BUDGET_SIZE);
                    svmPrintString(text);
                }

                // now print the common parameters
```

```
                sprintf(text, "Lambda regularization param. \t: %f\n", (*param).LAMBDA_PARAM);
                svmPrintString(text);
                switch ((*param).KERNEL)
                {
                    case KERNEL_FUNC_GAUSSIAN:
                        svmPrintString("Gaussian kernel used \t\t: K(x, y) = exp(-0.5 * gamma * ||x - y||^2)\n");
                        if ((*param).KERNEL_GAMMA_PARAM != 0.0)
                        {
                            sprintf(text, "Kernel width gamma \t\t: %f\n\n", (*param).KERNEL_GAMMA_PARAM);
                            svmPrintString(text);
                        }
                        else
                            svmPrintString("Kernel width gamma \t\t: 1 / DIMENSIONALITY\n\n");
                        break;

                    case KERNEL_FUNC_EXPONENTIAL:
                        svmPrintString("Exponential kernel used \t: K(x, y) = exp(-0.5 * gamma * ||x - y||)\n");
                        if ((*param).KERNEL_GAMMA_PARAM != 0.0)
                        {
                            sprintf(text, "Kernel width gamma \t\t: %f\n\n", (*param).KERNEL_GAMMA_PARAM);
                            svmPrintString(text);
                        }
                        else
                            svmPrintString("Kernel width gamma \t\t: 1 / DIMENSIONALITY\n\n");
                        break;

                    case KERNEL_FUNC_POLYNOMIAL:
                        sprintf(text, "Polynomial kernel used \t\t: K(x, y) = (x^T * y + %.2f)^%.2f\n\n", (*param).KERNEL_COEF_PARAM, (*param).KERNEL_DEGREE_PARAM);
                        svmPrintString(text);
                        break;

                    case KERNEL_FUNC_SIGMOID:
                        sprintf(text, "Sigmoid kernel used \t\t: K(x, y) = tanh(%.2f * x^T * y + %.2f)\n\n", (*param).KERNEL_DEGREE_PARAM, (*param).KERNEL_COEF_PARAM);
                        svmPrintString(text);
                        break;

                    case KERNEL_FUNC_LINEAR:
                        svmPrintString("Linear kernel used \t\t: K(x, y) = (x^T * y)\n\n");
                        break;

                    case KERNEL_FUNC_USER_DEFINED:
                        svmPrintString("User-defined kernel function used.\n\n");
                        break;
                }
            }
        }

        // increase dimensionality if bias term included
        if ((*param).BIAS_TERM != 0.0)
            (*param).DIMENSION++;

        // set gamma to default value of inverse dimensionality if not specified by a user
        if ((*param).KERNEL_GAMMA_PARAM == 0.0)
            (*param).KERNEL_GAMMA_PARAM = 1.0 / (*param).DIMENSION;
    }
    else
    {
        if (i >= argc - 2)
        {
            svmPrintErrorString("Error, input format not recognized. Run 'budgetedsvm-predict' for help.\n");
        }

        pFile = fopen(argv[i], "r");
        if (pFile == NULL)
        {
            sprintf(text, "Can't open input file %s!\n", argv[i]);
            svmPrintErrorString(text);
        }
        else
        {
            fclose(pFile);
            strcpy(inputFile, argv[i]);
        }

        pFile = fopen(argv[i + 1], "r");
        if (pFile == NULL)
        {
            sprintf(text, "Can't open model file %s!\n", argv[i + 1]);
            svmPrintErrorString(text);
        }
```

```
        else
        {
            fclose(pFile);
            strcpy(modelFile, argv[i + 1]);
        }

        pFile = fopen(argv[i + 2], "w");
        if (pFile == NULL)
        {
            sprintf(text, "Can't create output file %s!\n", argv[i + 2]);
            svmPrintErrorString(text);
        }
        else
        {
            fclose(pFile);
            strcpy(outputFile, argv[i + 2]);
        }

        // modify parameters
        for (unsigned int i = 0; i < option.size(); i++)
        {
            switch (option[i])
            {
                case 'v':
                    (*param).VERBOSE = (value[i] != 0);
                    break;

                case 'z':
                    (*param).CHUNK_SIZE = (unsigned int) value[i];
                    if ((*param).CHUNK_SIZE < 1)
                    {
                        sprintf(text, "Input parameter '-z' should be a positive real number!\nRun 'budgetedsvm-predict' for help.\n");
                        svmPrintErrorString(text);
                    }
                    break;

                case 'w':
                    (*param).CHUNK_WEIGHT = (unsigned int) value[i];
                    if ((*param).CHUNK_WEIGHT < 1)
                    {
                        sprintf(text, "Input parameter '-w' should be a positive real number!\nRun 'budgetedsvm-predict' for help.\n");
                        svmPrintErrorString(text);
                    }
                    break;

                case 'S':
                    (*param).VERY_SPARSE_DATA = (unsigned int) (value[i] != 0);
                    break;

                case 'o':
                    (*param).OUTPUT_SCORES = (value[i] != 0);
                    break;

                default:
                    sprintf(text, "Error, unknown input parameter '-%c'!\nRun 'budgetedsvm-predict' for help.\n", option[i]);
                    svmPrintErrorString(text);
                    break;
            }
        }

        // shut down printing to screen if user specified so
        if (!(*param).VERBOSE)
            setPrintStringFunction(NULL);
    }
}
```

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	*trainingPhase*	Indicator if training or testing phase instructions.
in	*param*	Parameter object modified by user input.

Definition at line 1187 of file budgetedSVM.cpp.

```
{
    char text[256];
    if (trainingPhase)
    {
        svmPrintString("\n Usage:\n");
        svmPrintString(" budgetedsvm-train [options] train_file [model_file]\n\n");
        svmPrintString(" Inputs:\n");
        svmPrintString(" options\t- parameters of the model\n");
        svmPrintString(" train_file\t- url of training file in LIBSVM format\n");
        svmPrintString(" model_file\t- file that will hold a learned model\n");
        svmPrintString(" --------------------------------------------\n");
        svmPrintString(" Options are specified in the following format:\n");
        svmPrintString(" '-OPTION1 VALUE1 -OPTION2 VALUE2 ...'\n\n");
        svmPrintString(" Following options are available; affected algorithm and default values are\n");
        svmPrintString("   given in parentheses (algorithm not specified if option affects all):\n\n");
        sprintf(text,  " A - algorithm, which large-scale SVM approximation to use (%d):\n", (*param).ALGORITHM);
        svmPrintString(text);
        svmPrintString("       0 - Pegasos\n");
        svmPrintString("       1 - AMM batch\n");
        svmPrintString("       2 - AMM online\n");
        svmPrintString("       3 - LLSVM\n");
        svmPrintString("       4 - BSGD\n");
        svmPrintString(" D - dimensionality (faster loading if set, if omitted inferred from the data)\n");
        svmPrintString(" B - limit on the number of weights per class in AMM, OR\n");
        sprintf(text, "       total SV set budget in BSGD, OR number of landmark points in LLSVM (%d)\n", (*param).BUDGET_SIZE);
        svmPrintString(text);
        sprintf(text, " L - lambda regularization parameter; high value -> less complex model (%.5f)\n", (*param).LAMBDA_PARAM);
        svmPrintString(text);
        sprintf(text, " b - bias term, if 0 no bias added (%.1f)\n", (*param).BIAS_TERM);
        svmPrintString(text);
        sprintf(text, " e - number of training epochs (AMM, BSGD; %d)\n", (*param).NUM_EPOCHS);
        svmPrintString(text);
        sprintf(text, " s - number of subepochs (AMM batch; %d)\n", (*param).NUM_SUBEPOCHS);
        svmPrintString(text);
        sprintf(text, " k - pruning frequency, after how many examples is pruning done (AMM; %d)\n", (*param).K_PARAM);
        svmPrintString(text);
        sprintf(text, " c - pruning threshold; high value -> less complex model (AMM; %.2f)\n", (*param).C_PARAM);
        svmPrintString(text);
        svmPrintString(" K - kernel function (0 - RBF; 1 - exponential, 2 - polynomial; 3 - linear, \n");
        sprintf(text, "       4 - sigmoid; 5 - user-defined) (LLSVM, BSGD; %d)\n", (*param).KERNEL);
        svmPrintString(text);
        sprintf(text, " g - RBF or exponential kernel width gamma (LLSVM, BSGD; 1/DIMENSIONALITY)\n");
        svmPrintString(text);
        sprintf(text, " d - polynomial kernel degree or sigmoid kernel slope (LLSVM, BSGD; %.2f)\n", (*param).KERNEL_DEGREE_PARAM);
        svmPrintString(text);
        sprintf(text, " i - polynomial or sigmoid kernel intercept (LLSVM, BSGD; %.2f)\n", (*param).KERNEL_COEF_PARAM);
        svmPrintString(text);
        svmPrintString(" m - budget maintenance in BSGD (0 - removal; 1 - merging, uses Gaussian kernel), OR\n");
        sprintf(text,  "       landmark selection in LLSVM (0 - random; 1 - k-means; 2 - k-medoids) (%d)\n", (*param).MAINTENANCE_SAMPLING_STRATEGY);
        svmPrintString(text);

        sprintf(text, " C - clone probability when misclassification occurs in AMM (%.2f)\n", (*param).CLONE_PROBABILITY);
        svmPrintString(text);
        sprintf(text, " y - clone probability decay when weight cloning occurs in AMM (%.2f)\n\n", (*param).CLONE_PROBABILITY_DECAY);
        svmPrintString(text);

        svmPrintString(" z - training and test file are loaded in chunks so that the algorithms can\n");
        svmPrintString("       handle budget files on weaker computers; z specifies number of examples\n");
        sprintf(text,  "       loaded in a single chunk of data (%d)\n", (*param).CHUNK_SIZE);
        svmPrintString(text);
        svmPrintString(" w - model weights are split in chunks, so that the algorithm can handle\n");
```

```
        svmPrintString("       highly dimensional data on weaker computers; w specifies number of\n");
        sprintf(text,  "       dimensions stored in one chunk (%d)\n", (*param).CHUNK_WEIGHT);
        svmPrintString(text);
        svmPrintString(" S - if set to 1 data is assumed sparse, if 0 data assumed non-sparse; used to\n");
        svmPrintString("       speed up kernel computations (default is 1 when percentage of non-zero\n");
        svmPrintString("       features is less than 5%, and 0 when percentage is larger than 5%)\n");
        sprintf(text, " r - randomize the algorithms; 1 to randomize, 0 not to randomize (%d)\n", (*param).RANDOMIZE);
        svmPrintString(text);
        sprintf(text, " v - verbose output; 1 to show the algorithm steps, 0 for quiet mode (%d)\n\n", (*param).VERBOSE);
        svmPrintString(text);
    }
    else
    {
        svmPrintString("\n Usage:\n");
        svmPrintString(" budgetedsvm-predict [options] test_file model_file output_file\n\n");
        svmPrintString(" Inputs:\n");
        svmPrintString(" options\t- parameters of the model\n");
        svmPrintString(" test_file\t- url of test file in LIBSVM format\n");
        svmPrintString(" model_file\t- file that holds a learned model\n");
        svmPrintString(" output_file\t- url of file where output will be written\n");
        svmPrintString(" ---------------------------------------------\n");
        svmPrintString(" Options are specified in the following format:\n");
        svmPrintString(" '-OPTION1 VALUE1 -OPTION2 VALUE2 ...'\n\n");
        svmPrintString(" The following options are available (default values in parentheses):\n\n");

        svmPrintString(" z - the training and test file are loaded in chunks so that the algorithm can\n");
        svmPrintString("       handle budget files on weaker computers; z specifies number of examples\n");
        sprintf(text,  "       loaded in a single chunk of data (%d)\n", (*param).CHUNK_SIZE);
        svmPrintString(text);
        svmPrintString(" w - the model weight is split in parts, so that the algorithm can handle\n");
        svmPrintString("       highly dimensional data on weaker computers; w specifies number of\n");
        sprintf(text,  "       dimensions stored in one chunk (%d)\n", (*param).CHUNK_WEIGHT);
        svmPrintString(text);
        svmPrintString(" S - if set to 1 data is assumed sparse, if 0 data assumed non-sparse, used to\n");
        svmPrintString("       speed up kernel computations (default is 1 when percentage of non-zero\n");
        svmPrintString("       features is less than 5%, and 0 when percentage is larger than 5%)\n");
        svmPrintString(" o - if set to 1, the output file will contain not only the class predictions,\n");
        sprintf(text,  "       but also tab-delimited scores of the winning class (%d)\n", (*param).OUTPUT_SCORES);
        svmPrintString(text);
        sprintf(text, " v - verbose output; 1 to show algorithm steps, 0 for quiet mode (%d)\n\n", (*param).VERBOSE);
        svmPrintString(text);
    }
}
```

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	*fileName*	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.

```
{
    FILE *pFile = NULL;
    if (fileName)
    {
        pFile = fopen(fileName, "r");
        if (pFile != NULL)
        {
            fclose(pFile);
            return true;
        }
    }
    return false;
}
```

6.3.2.5 setPrintErrorStringFunction()

```
void setPrintErrorStringFunction (
            funcPtr printFunc )
```

Modifies a callback that prints an error string.

Parameters

in	*printFunc*	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.

```
{
    svmPrintErrorStringStatic = (printFunc == NULL) ? &printErrorDefault : printFunc;
}
```

6.3.2.6 setPrintStringFunction()

```
void setPrintStringFunction (
            funcPtr printFunc )
```

Modifies a callback that prints a string.

Parameters

in	*printFunc*	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.

```
{
    if (printFunc == NULL)
        svmPrintStringStatic = &printNull;
    else
        svmPrintStringStatic = printFunc;
}
```

6.3.2.7 svmPrintErrorString()

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

Prints error string to the output.

Parameters

in	*text*	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 convinient 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.

```
{
    svmPrintErrorStringStatic(text);
}
```

6.3.2.8 svmPrintString()

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

Prints string to the output.

Parameters

in	*text*	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 convinient 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.

```
{
    svmPrintStringStatic(text);
}
```

6.4 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSV↩ M/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	*testData*	Input test data.
in	*param*	The parameters of the algorithm.
in	*model*	Trained LLSVM model.
out	*labels*	Vector of predicted labels.
out	*scores*	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, outputing the predicted labels and the error rate.

Definition at line 362 of file llsvm.cpp.

```
{
    // to train linear kernel we need -1 and +1 labels, but a user can give us any labels, e.g., 0/1 labels; therefor
    //  here we set the default labels, such that the first user-provided label is renamed as -1, and the second +1
    int defaultLabels[2] = {-1, 1};

    unsigned long N, err = 0, total = 0, timeCalc = 0, start;
    bool stillChunksLeft = true;
    char text[256];
    VectorXd v((*param).BUDGET_SIZE), temp((*param).BUDGET_SIZE);
    budgetedVectorLLSVM *currentData = NULL;
    long double tempSqrNorm;

    if ((*param).VERBOSE)
        svmPrintString("Computing lower-dimensional representation and predicting labels ...\n");

    while (stillChunksLeft)
    {
        stillChunksLeft = testData->readChunk((*param).CHUNK_SIZE);
        (*param).updateVerySparseDataParameter(testData->getSparsity());

        N = testData->N;
        total += N;
        start = clock();

        // calculate E, kernel between testing points and landmark points
        VectorXd predictions(N);
        for (unsigned int i = 0; i < N; i++)
        {
            if ((*param).VERY_SPARSE_DATA)
            {
                // since we are computing kernels using vectors directly from the budgetedData, we need square norm of the vector to speed-up
                //  computations, here we compute it just once; no need to do it in non-sparse case, since this norm can be retrieved directly
                //  from budgetedVector

                tempSqrNorm = testData->getVectorSqrL2Norm(i, param);
                for (unsigned int j = 0; j < (*param).BUDGET_SIZE; j++)
                {
                    v(j) = (double)(*(model->modelLLSVMlandmarks))[j]->computeKernel(i, testData, param, tempSqrNorm);
                }
            }
            else
            {
                // first create the budgetedVector using the vector from budgetedData, to be used in gaussianKernel() method below
                currentData = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                currentData->createVectorUsingDataPoint(testData, i, param);

                for (unsigned int j = 0; j < (*param).BUDGET_SIZE; j++)
                {
                    v(j) = (double)(*(model->modelLLSVMlandmarks))[j]->computeKernel(currentData, param);
                }
                delete currentData;
                currentData = NULL;
            }

            temp = v.transpose() * (*model).modelLLSVMmatrixW;
            predictions(i) = temp.dot((*model).modelLLSVMweightVector);
        }

        for (unsigned int i = 0; i < N; i++)
        {
            if ((predictions(i) > 0.0) != (defaultLabels[testData->al[i]] > 0))
                err++;

            // save predicted label, will be sent to output ...
            if (labels)
                (*labels).push_back((int)(testData->yLabels)[(predictions(i) > 0)]);
            // ... and the scores
            if (scores)
                (*scores).push_back((float) abs(predictions(i)));
```

```
        }

        timeCalc += clock() - start;

        if (((*param).VERBOSE) && (N > 0))
        {
            sprintf(text, "Number of examples processed: %ld\n", total);
            svmPrintString(text);
        }
    }

    if ((*param).VERBOSE)
    {
        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);
        svmPrintString(text);
    }

    return (float) (100.0 * (float) err / (float) total);
}
```

6.4.2.2 trainLLSVM()

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

Train LLSVM online.

Parameters

in	*trainData*	Input training data.
in	*param*	The parameters of the algorithm.
in,out	*model*	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.

```
{
    unsigned long timeCalc = 0, start;
    unsigned int i, j, total = 0, N, temp;
    bool stillChunksLeft = true, firstChunk = true;
    long double tempSqrNorm;
    char text[256];
    budgetedVectorLLSVM *currentData = NULL;

    // W matrix for Nystrom method, here employ Eigen library since we need complex matrix operations
    (*model).modelLLSVMmatrixW = MatrixXd::Zero((*param).BUDGET_SIZE, (*param).BUDGET_SIZE);

    // initialize weight (i.e., hyperplane) in the projected space to zero-vector
    (*model).modelLLSVMweightVector = VectorXd::Zero((*param).BUDGET_SIZE);

    // commence with LLSVM training procedure
    stillChunksLeft = true;
    while (stillChunksLeft)
    {
        stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);

        // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
        //  (of course, in the case of AMM, speeds up linear kernel computation)
        (*param).updateVerySparseDataParameter(trainData->getSparsity());

        // compute observed data dimensionality, where we also account for possible bias term, and check if
        //  we need to expand the current model weights if some new data dimensions were found during loading
        temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
```

```
        if ((*param).DIMENSION < temp)
        {
            /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION, temp);
            svmPrintString(text);*/
            (*model).extendDimensionalityOfModel(temp, param);

            // update the dimensionality
            (*param).DIMENSION = temp;
        }

        if (trainData->yLabels.size() != 2)
        {
            sprintf(text, "LLSVM is a binary classifier, but %d class(es) detected!\n", (int) trainData->yLabels.size());
            svmPrintErrorString(text);
        }

        N = trainData->N;
        total += N;
        start = clock();

        // if we just started training initialize the landmark points
        if (firstChunk)
        {
            if (N < (*param).BUDGET_SIZE)
            {
                trainData->flushData();
                svmPrintErrorString("Number of landmark points larger than size of the loaded chunk!\n");
            }
            firstChunk = false;

            // select landmark points
            selectLandmarkPoints(trainData, param, model);

            if ((*param).VERBOSE)
                svmPrintString("Computing the mapping function ...\n");

            // compute the W matrix, done just once per training
            for (i = 0; i < (*param).BUDGET_SIZE; i++)
            {
                for (j = 0; j < (*param).BUDGET_SIZE; j++)
                {
                    if (i <= j)
                    {
                        (*model).modelLLSVMmatrixW(i, j) = (double) (*(model->modelLLSVMlandmarks))[i]->computeKernel((*(model->modelLLSVMlandmarks))[j], param);
                    }
                    else
                    {
                        (*model).modelLLSVMmatrixW(i, j) = (*model).modelLLSVMmatrixW(j, i);
                    }
                }
            }

            // finally, compute K_zz = W^(-0.5), initialization is complete
            invSquareRoot((*model).modelLLSVMmatrixW);
        }

        // done with initialization phase, next we compute the mapping in the new space and solve linear SVM

        if ((*param).VERBOSE)
            svmPrintString("Computing mapping of the training data ...\n");

        // here compute kernel matrix E between input data and landmark points
        MatrixXd E = MatrixXd::Zero(N, (*param).BUDGET_SIZE);
        for (i = 0; i < N; i++)
        {
            if ((*param).VERY_SPARSE_DATA)
            {
                // since we are computing kernels using vectors directly from the budgetedData, we need square norm of the vector to speed-up
                //  computations, here we compute it just once; no need to do it in non-sparse case, since this norm can be retrieved directly
                //  from budgetedVector
                tempSqrNorm = trainData->getVectorSqrL2Norm(i, param);
                for (j = 0; j < (*param).BUDGET_SIZE; j++)
                {
                    E(i, j) = (double)(*(model->modelLLSVMlandmarks))[j]->computeKernel(i, trainData, param, tempSqrNorm);
                }
            }
            else
            {
                // first create the budgetedVector using the vector from budgetedData, to be used in gaussianKernel() method below
```

```
                currentData = new budgetedVectorLLSVM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                currentData->createVectorUsingDataPoint(trainData, i, param);

                for (j = 0; j < (*param).BUDGET_SIZE; j++)
                {
                    E(i, j) = (double)(*(model->modelLLSVMlandmarks))[j]->computeKernel(currentData,
       param);
                }
                delete currentData;
                currentData = NULL;
            }
        }

        // compute new representation of data set which will be used to train SVM
        E = E * (*model).modelLLSVMmatrixW;

        if ((*param).VERBOSE)
            svmPrintString("Training linear SVM ...\n");

        // now we can move on to training SVM using data in a new space
        liblinear_Solve_l2r_l1(E, trainData->al, (*model).modelLLSVMweightVector, param,
       &(trainData->yLabels));
        timeCalc += clock() - start;

        if (((*param).VERBOSE) && (N > 0))
        {
            sprintf(text, "Number of examples processed: %d\n", total);
            svmPrintString(text);
        }
    }
    // training done, get rid of training data
    trainData->flushData();

    //timeCalc += clock() - startTotal;
    if ((*param).VERBOSE)
    {
        sprintf(text, "*** Training completed in %5.3f seconds.\n\n", (double)timeCalc /
       (double)CLOCKS_PER_SEC);
        svmPrintString(text);
    }
}
```

6.5 C:/Users/Nemanja/Documents/Yahoo/Downloads/BudgetedSV↩M/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	*testData*	Input test data.
in	*param*	The parameters of the algorithm.
in	*model*	Trained MM model.
out	*labels*	Vector of predicted labels.
out	*scores*	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, outputing the predicted labels and the error rate.

Definition at line 358 of file mm_algs.cpp.

```
{
    unsigned long N, err = 0, totalPoints = 0;
    long double fx, maxFx;
    bool stillChunksLeft = true;
    long start, timeCalc = 0;
    char text[1024];
    budgetedVectorAMM *currentData = NULL;
    long double *classMaxScores = new long double[(testData->yLabels).size()];

    while (stillChunksLeft)
    {
        stillChunksLeft = testData->readChunk((*param).CHUNK_SIZE);
        (*param).updateVerySparseDataParameter(testData->getSparsity());

        N = testData->N;
        start = clock();
        for (unsigned int r = 0; r < N; r++)
        {
            totalPoints++;
            unsigned int y = 0;
            maxFx = -INF;

            if ((*param).VERY_SPARSE_DATA)
            {
                // compute kernels using vectors directly from the budgetedData
                for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
                {
                    classMaxScores[i] = -INF;
                    for (unsigned int j = 0; j < (*(model->getModel()))[i].size(); j++)
                    {
                        fx = (*(model->getModel()))[i][j]->linearKernel(r, testData, param);
                        if (fx > maxFx)
                        {
                            maxFx = fx;
                            y = i;
                        }

                        if (fx > classMaxScores[i])
                            classMaxScores[i] = fx;
                    }
                }
            }
            else
            {
                // first create the budgetedVector using the vector from budgetedData, to be used in gaussianKernel() method below
                currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                currentData->budgetedVector::createVectorUsingDataPoint(testData, r, param);

                for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
                {
                    classMaxScores[i] = -INF;
                    for (unsigned int j = 0; j < (*(model->getModel()))[i].size(); j++)
                    {
                        fx = (*(model->getModel()))[i][j]->linearKernel(currentData);
                        if (fx > maxFx)
                        {
                            maxFx = fx;
                            y = i;
                        }

                        if (fx > classMaxScores[i])
                            classMaxScores[i] = fx;
                    }
                }
                delete currentData;
                currentData = NULL;
            }

            // save predicted label, will be sent to output ...
            if (labels)
                (*labels).push_back((int)(testData->yLabels)[y]);
            // ... and the scores
            if (scores)
            {
                // for AMM models the score is the difference between winning and the second best score
                long double secondBestScore = -INF;
                for (unsigned int i = 0; i < (testData->yLabels).size(); i++)
                {
                    if (i == y)
                        continue;

                    if (secondBestScore < classMaxScores[i])
                        secondBestScore = classMaxScores[i];
                }
                (*scores).push_back((float) (maxFx - secondBestScore));
            }
```

```
            if (y != testData->al[r])
                err++;
        }

        timeCalc += clock() - start;

        if (((*param).VERBOSE) && (N > 0))
        {
            sprintf(text, "Number of examples processed: %ld\n", totalPoints);
            svmPrintString(text);
        }
    }
    delete[] classMaxScores;
    testData->flushData();

    if ((*param).VERBOSE)
    {
        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);
        svmPrintString(text);
    }

    return (float) (100.0 * (float)err / (float)totalPoints);
}
```

6.5.2.2 trainAMMbatch()

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

Train AMM batch.

Parameters

in	*trainData*	Input training data.
in	*param*	The parameters of the algorithm.
in,out	*model*	Initial AMM model.

The function trains multi-hyperplane machine using AMM batch algotihm, 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.

```
{
    vector <unsigned int> n;    // stores number of weights per class
    unsigned long timeCalc = 0, start;
    long double fx1, fx2, maxFx, assocFx;
    unsigned int i, j, t, N, i1, i2, j1, j2, sizeOfyLabels = 0, countNew = 0, countDel = 0, numIter = 0, currAssign = 0, currAssignID, temp;
    bool stillChunksLeft;
    char text[1024];
    budgetedVectorAMM *currentData = NULL;

    //Initialization phase with algorithm AMM_online
    stillChunksLeft = true;
    while (stillChunksLeft)
    {
        stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);

        // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
        //  (of course, in the case of AMM, speeds up linear kernel computation)
        (*param).updateVerySparseDataParameter(trainData->getSparsity());

        // compute observed data dimensionality, where we also account for possible bias term, and check if
        //  we need to expand the current model weights if some new data dimensions were found during loading
        temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
```

```
        if ((*param).DIMENSION < temp)
        {
            /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION, temp);
            svmPrintString(text);*/
            (*model).extendDimensionalityOfModel(temp, param);

            // update the dimensionality
            (*param).DIMENSION = temp;
        }

        N = trainData->N;
        if (numIter == 0)
        {
            //Initialize
            sizeOfyLabels = (unsigned int) trainData->yLabels.size();
            for (i = 0; i < sizeOfyLabels; i++)
            {
                 n.push_back(1);

                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_SIZE);
                 vector <budgetedVectorAMM*> perClassWeights;
                 perClassWeights.push_back(currentData);
                 currentData = NULL;
                 (*((*model).getModel())).push_back(perClassWeights);
             }
         }
         else if (sizeOfyLabels != (unsigned int) trainData->yLabels.size())
         {
             // if in previous chunks some class wasn't observed, could happen with small chunks or unbalanced classes
             // just add new zero weights for the new classes
             for (i = 0; i < (trainData->yLabels.size() - sizeOfyLabels); i++)
             {
                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                 vector <budgetedVectorAMM*> perClassWeights;
                 perClassWeights.push_back(currentData);
                 currentData = NULL;
                 (*((*model).getModel())).push_back(perClassWeights);
             }
             sizeOfyLabels = (unsigned int) trainData->yLabels.size();
         }

         // randomize
         vector <unsigned int> tv(N, 0);
         unsigned int *assigns = new unsigned int[N];
         for (i = 0; i < N; i++)
         {
             tv[i] = i;
         }
         if ((*param).RANDOMIZE)
             random_shuffle(tv.begin(), tv.end());

         start = clock();
         for (unsigned int trainIter = 0; trainIter < N; trainIter++)
         {
             numIter++;
             t = tv[trainIter];

             if (!(*param).VERY_SPARSE_DATA)
             {
                 // only create currentData if the data is non-sparse, otherwise kernels will be computed directly from trainData
                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                 currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
             }

             // calculate i+, j+
             i1 = trainData->al[t];
             j1 = 0;
             maxFx = -INF;
             for (j = 0; j < n[i1]; j++)
             {
                 if ((*param).VERY_SPARSE_DATA)
                     fx1 = (*((*model).getModel()))[i1][j]->linearKernel(t, trainData, param);
                 else
                     fx1 = (*((*model).getModel()))[i1][j]->linearKernel(currentData);

                 if (fx1 > maxFx)
                 {
                     j1 = j;
                     maxFx = fx1;
                 }
             }
             fx1 = maxFx;
             *(assigns + t) = (*((*model).getModel()))[i1][j1]->getID();

             // calculate i-, j-
```

```
            i2 = 0;
            j2 = 0;
            fx2 = 0;
            maxFx = -INF;
            for (i = 0; i < sizeOfyLabels; i++)
            {
                if (i == i1)
                    continue;

                for (j = 0; j < n[i]; j++)
                {
                    if ((*param).VERY_SPARSE_DATA)
                        fx2 = (*((*model).getModel()))[i][j]->linearKernel(t, trainData, param);
                    else
                        fx2 = (*((*model).getModel()))[i][j]->linearKernel(currentData);

                    if (fx2 > maxFx)
                    {
                        maxFx = fx2;
                        i2 = i;
                        j2 = j;
                    }
                }
            }
            fx2 = maxFx;

            // downgrade weight each iteration
            for (i = 0; i < sizeOfyLabels; i++)
                for (j = 0; j < n[i]; j++)
                    (*((*model).getModel()))[i][j]->downgrade(numIter);

            if (1.0 + fx2 - fx1 > 0.0)
            {
                // we made a misprediction, push negative class further away, and positive closer!
                if ((*param).VERY_SPARSE_DATA)
                {
                    // since we did not create currentData earlier, here we create it to perform updates
                    currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                    currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
                }

                // push the other class further away
                (*((*model).getModel()))[i2][j2]->updateUsingVector(currentData, numIter, -1, param);

                // update the true class weight
                if (fx1 > 0.0)
                {
                    // here clone the best weight if the cloning probability allows it
                    if ((unsigned int) n[i1] < (*param).BUDGET_SIZE)
                    {
                        if ((*param).CLONE_PROBABILITY > get_random_probability())
                        {
                            // clone the winning weight
                            budgetedVectorAMM *clonedVector = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                            clonedVector->createVectorUsingVector((*((*model).getModel()))[i1][j1]);

                            // add the new cloned weight to the model
                            (*((*model).getModel()))[i1].push_back(clonedVector);
                            n[i1]++;
                            clonedVector = NULL;

                            // update the clone probability after successful cloning
                            (*param).CLONE_PROBABILITY *= (*param).CLONE_PROBABILITY_DECAY;
                        }
                    }

                    (*((*model).getModel()))[i1][j1]->updateUsingVector(currentData, numIter, 1, param);

                    delete currentData;
                    currentData = NULL;
                }
                else
                {
                    if ((unsigned int) n[i1] < (*param).BUDGET_SIZE)
                    {
                        n[i1]++;
                        currentData->updateDegradation(numIter, param);
                        (*((*model).getModel()))[i1].push_back(currentData);
                        currentData = NULL;
                        countNew++;
                    }
                    else
                    {
                        delete currentData;
```

```
                      currentData = NULL;
                  }
              }
          }
          else
          {
              if (!(*param).VERY_SPARSE_DATA)
              {
                  // if sparse data then no need for this part, since we didn't even create currentData
                  delete currentData;
                  currentData = NULL;
              }
          }
      }
      timeCalc += clock() - start;

      trainData->saveAssignment(assigns);
      delete [] assigns;

      if (((*param).VERBOSE) && (N > 0))
      {
          sprintf(text, "Number of examples processed: %d\n", numIter);
          svmPrintString(text);
      }
  }

  if ((*param).VERBOSE)
      svmPrintString("Initialization epoch done!\n");

  // end of init phase, start AMM algorithm below

  for (unsigned int epoch = 1; epoch <= (*param).NUM_EPOCHS; epoch++)
  {
      stillChunksLeft = true;
      while (stillChunksLeft)
      {
          stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE, true);

          // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
          //  (of course, in the case of AMM, speeds up linear kernel computation)
          (*param).updateVerySparseDataParameter(trainData->getSparsity());

          // compute observed data dimensionality, where we also account for possible bias term, and check if
          //  we need to expand the current model weights if some new data dimensions were found during loading
          temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
          if ((*param).DIMENSION < temp)
          {
              /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION, temp);
              svmPrintString(text);*/
              (*model).extendDimensionalityOfModel(temp, param);

              // update the dimensionality
              (*param).DIMENSION = temp;
          }

          N = trainData->N;
          trainData->readChunkAssignments(!stillChunksLeft);

          // randomize
          vector <int> tv(N, 0);
          for (unsigned int ti = 0; ti < N; ti++)
              tv[ti] = ti;

          if ((*param).RANDOMIZE)
              random_shuffle(tv.begin(), tv.end());

          start = clock();
          for (unsigned int trainIter = 0; trainIter < N; trainIter++)
          {
              numIter++;
              t = tv[trainIter];

              if (!(*param).VERY_SPARSE_DATA)
              {
                  currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                  currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
              }

              // calculate i+, j+
              i1 = trainData->al[t];
              j1 = 0;

              currAssignID = trainData->assignments[t];
```

```
                maxFx = 0;
                assocFx = -INF;
                for (j = 0; j < n[i1]; j++)
                {
                    if ((*param).VERY_SPARSE_DATA)
                        fx1 = (*((*model).getModel()))[i1][j]->linearKernel(t, trainData, param);
                    else
                        fx1 = (*((*model).getModel()))[i1][j]->linearKernel(currentData);

                    if ((maxFx == 0) || (fx1 > maxFx))
                    {
                        j1 = j;
                        maxFx = fx1;
                    }

                    // this is the prediction of the associated same-label weight
                    if ((*((*model).getModel()))[i1][j]->getID() == currAssignID)
                    {
                       currAssign = j;
                       assocFx = fx1;
                    }
                }

                fx1 = maxFx;
                if (assocFx == -INF)
                {
                    assocFx = maxFx;
                    currAssign = j1;
                }

                // calculate i-, j-
                i2 = 0;
                j2 = 0;
                fx2 = 0;
                maxFx = 0;
                for (i = 0; i < sizeOfyLabels; i++)
                {
                    if (i == i1)
                        continue;

                    for (j = 0; j < n[i]; j++)
                    {
                        if ((*param).VERY_SPARSE_DATA)
                            fx2 = (*((*model).getModel()))[i][j]->linearKernel(t, trainData, param);
                        else
                            fx2 = (*((*model).getModel()))[i][j]->linearKernel(currentData);

                        if ((maxFx == 0) || (fx2 > maxFx))
                        {
                            maxFx = fx2;
                            i2 = i;
                            j2 = j;
                        }
                    }
                }
                fx2 = maxFx;

                // downgrade weights each iteration
                for (unsigned int i = 0; i < sizeOfyLabels; i++)
                {
                    for (unsigned int j = 0; j < n[i]; j++)
                    {
                        (*((*model).getModel()))[i][j]->downgrade(numIter);
                    }
                }

                // calculate v
                if (1.0 + fx2 - assocFx > 0.0)
                {
                    // we made a misprediction, update the weights by pushing the wrong-class weight further from the misclassified
                    //  example, and the true-class closer to the misclassified example
                    if ((*param).VERY_SPARSE_DATA)
                    {
                        // since we did not create currentData earlier, here we create it to perform updates
                        currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                        currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
                    }

                    // duplicate the assigned true-class weight if the cloning probability allows it
                    if ((assocFx > 0.0) && ((unsigned int)n[i1] < (*param).BUDGET_SIZE))
                    {
                        if ((*param).CLONE_PROBABILITY > get_random_probability())
                        {
                            // clone the associated weight
```

```
                            budgetedVectorAMM *clonedVector = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                            clonedVector->createVectorUsingVector((*((*model).getModel()))[i1][currAssign]);

                            // add the new cloned weight to the model
                            (*((*model).getModel()))[i1].push_back(clonedVector);
                            n[i1]++;
                            clonedVector = NULL;

                            // update the clone probability after successful cloning
                            (*param).CLONE_PROBABILITY *= (*param).CLONE_PROBABILITY_DECAY;
                        }
                    }

                    (*((*model).getModel()))[i1][currAssign]->updateUsingVector(currentData, numIter, 1, param);
                    (*((*model).getModel()))[i2][j2]->updateUsingVector(currentData, numIter, -1, param);
                    if ((fx1 <= 0.0) && (n[i1] < (*param).BUDGET_SIZE))
                    {
                        n[i1]++;
                        currentData->updateDegradation(numIter, param);
                        (*((*model).getModel()))[i1].push_back(currentData);
                        currentData = NULL;
                        countNew++;
                    }
                    else
                    {
                        // if over the budget, we do not add a new data point to the budget
                        delete currentData;
                        currentData = NULL;
                    }
                }
                else
                {
                    if (!(*param).VERY_SPARSE_DATA)
                    {
                        // if sparse data then no need for this part, since we didn't even create currentData
                        delete currentData;
                        currentData = NULL;
                    }
                }

                timeCalc += clock() - start;
                start = clock();

                if (numIter % (*param).K_PARAM == 0)
                {
                    // we run the pruning procedure here
                    long double sumNorms = 0;
                    long double sumThreshold = (long double)(*param).C_PARAM * (long double)(*param).C_PARAM / ((long double)numIter * (long double)numIter * (*param).LAMBDA_PARAM * (*param).LAMBDA_PARAM);
                    vector <long double> weightNorms, sortedWeightNorms;
                    int numToDelete = 0;

                    // first find the norms of weights
                    for (unsigned int i = 0; i < sizeOfyLabels; i++)
                    {
                        for (vector<budgetedVectorAMM*>::iterator vi = (*((*model).getModel()))[i].begin(); vi != (*((*model).getModel()))[i].end(); vi++)
                        {
                            weightNorms.push_back((double) (*(*vi)).getSqrL2norm());
                        }
                    }

                    // now sort them
                    sortedWeightNorms = weightNorms;
                    sort(sortedWeightNorms.begin(), sortedWeightNorms.end());

                    // find how many before threshold is exceeded
                    for (unsigned int i = 0; i < weightNorms.size(); i++)
                    {
                        sumNorms += sortedWeightNorms[i];
                        if (sumNorms > sumThreshold)
                           break;
                        else
                           numToDelete++;
                    }

                    // delete those that should be deleted, with aggregate norm less than the set threshold
                    int counter = 0;
                    bool deleted = false;
                    for (unsigned int i = 0; i < sizeOfyLabels; i++)
```

```
                    {
                        for (vector<budgetedVectorAMM*>::iterator vi =
    (*((*model).getModel()))[i].begin(); vi != (*((*model).getModel()))[i].end();)
                        {
                            long double currNorm = weightNorms[counter++];

                            deleted = false;
                            for (int j = 0; j < numToDelete; j++)
                            {
                                if (currNorm == sortedWeightNorms[j])
                                {
                                    if (n[i] == 1)
                                    {
                                        svmPrintString("Was about to delete all weights of a class,
    check the K_PARAM and C_PARAM parameters!\n");
                                        break;
                                    }

                                    delete (*vi);
                                    vi = (*((*model).getModel()))[i].erase(vi);
                                    n[i]--;

                                    countDel++;
                                    deleted = true;
                                    break;
                                }
                            }

                            if (!deleted)
                               vi++;
                        }
                    }
                }
            }
            timeCalc += clock() - start;

            if (((*param).VERBOSE) && (N > 0))
            {
                sprintf(text, "Number of examples processed: %d\n", numIter);
                svmPrintString(text);
            }
        }

        // every so-so (around 3) subepochs recalculate the associations
        if (((epoch % (*param).NUM_SUBEPOCHS) == 0) && (epoch != (*param).NUM_EPOCHS))
        {
            // calculate the new assignments
            stillChunksLeft = true;
            while (stillChunksLeft)
            {
                stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);
                N = trainData->N;
                unsigned int *assigns = new unsigned int[N];

                start = clock();
                for (unsigned int ot = 0; ot < N; ot++)
                {
                    t = ot;
                    if ((*param).VERY_SPARSE_DATA)
                    {
                        // calculate i+, j+
                        i1 = trainData->al[t];
                        j1 = 0;
                        maxFx = -INF;
                        for (unsigned int j = 0; j < n[i1]; j++)
                        {
                            fx1 = (*((*model).getModel()))[i1][j]->linearKernel(t, trainData, param);
                            if (fx1 > maxFx)
                            {
                                j1 = j;
                                maxFx = fx1;
                            }
                        }
                    }
                    else
                    {
                        currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                        currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);

                        // calculate i+, j+
                        i1 = trainData->al[t];
                        j1 = 0;
                        maxFx = -INF;
                        for (unsigned int j = 0; j < n[i1]; j++)
                        {
                            fx1 = (*((*model).getModel()))[i1][j]->linearKernel(currentData);
                            if (fx1 > maxFx)
```

```
                            {
                                j1 = j;
                                maxFx = fx1;
                            }
                        }
                        delete currentData;
                        currentData = NULL;
                    }

                    *(assigns + t) = (*((*model).getModel()))[i1][j1]->getID();
                }
                timeCalc += clock() - start;

                trainData->saveAssignment(assigns);
                delete [] assigns;
            }
        }

        if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
        {
            sprintf(text, "Epoch %d/%d done.\n", epoch, (*param).NUM_EPOCHS);
            svmPrintString(text);
        }
    }
    trainData->flushData();

    if ((*param).VERBOSE)
    {
        sprintf(text, "*** Training completed in %5.3f seconds.\nNumber of weights deleted: %d\n", (double) timeCalc / (double) CLOCKS_PER_SEC, countDel);
        svmPrintString(text);
        for (unsigned int i = 0; i < sizeOfyLabels; i++)
        {
            sprintf(text, "Number of weights of class %d: %d\n", i + 1, n[i]);
            svmPrintString(text);
        }
    }
}
```

6.5.2.3 trainAMMonline()

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

Train AMM online.

Parameters

in	*trainData*	Input training data.
in	*param*	The parameters of the algorithm.
in,out	*model*	Initial AMM model.

The function trains multi-hyperplane machine using AMM online algotihm, 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++)
```

```
 {
     stillChunksLeft = true;
     while (stillChunksLeft)
     {
         stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);

         // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
         //  (of course, in the case of AMM, speeds up linear kernel computation)
         (*param).updateVerySparseDataParameter(trainData->getSparsity());

         // compute observed data dimensionality, where we also account for possible bias term, and check if
         //  we need to expand the current model weights if some new data dimensions were found during loading
         temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
         if ((*param).DIMENSION < temp)
         {
             /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION, temp);
             svmPrintString(text);*/
             (*model).extendDimensionalityOfModel(temp, param);

             // update the dimensionality
             (*param).DIMENSION = temp;
         }

         N = trainData->N;
         if (numIter == 0)
         {
             // initialize the model with zero weights
             sizeOfyLabels = (unsigned int) trainData->yLabels.size();
             for (unsigned int i = 0; i < sizeOfyLabels; i++)
             {
                 n.push_back(1);

                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                 vector <budgetedVectorAMM*> v1;
                 v1.push_back(currentData);
                 currentData = NULL;
                 (*((*model).getModel())).push_back(v1);
             }
         }
         else if (sizeOfyLabels != trainData->yLabels.size())
         {
             // if in the chunks before some class wasn't observed, could happen with small chunks or unbalanced classes
             for (unsigned int i = 0; i < (trainData->yLabels.size() - sizeOfyLabels); i++)
             {
                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                 vector <budgetedVectorAMM*> perClassWeights;
                 perClassWeights.push_back(currentData);
                 currentData = NULL;
                 (*((*model).getModel())).push_back(perClassWeights);
             }
             sizeOfyLabels = (unsigned int) trainData->yLabels.size();
         }

         // randomize
         vector <unsigned int> tv(N, 0);
         for (unsigned int ti = 0; ti < N; ti++)
         {
             tv[ti] = ti;
         }
         if ((*param).RANDOMIZE)
             random_shuffle(tv.begin(), tv.end());

         start = clock();
         for (unsigned int ot = 0; ot < N; ot++)
         {
             numIter++;
             t = tv[ot];

             if (!(*param).VERY_SPARSE_DATA)
             {
                 // only create currentData if the data is non-sparse, otherwise kernels will be computed directly from trainData
                 currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                 currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
             }

             //calculate i+,j+
             i1 = trainData->al[t];
             j1 = 0;
             maxFx = -INF;

             for (unsigned int j = 0; j < n[i1]; j++)
             {
```

```
    if ((*param).VERY_SPARSE_DATA)
        fx1 = (*((*model).getModel()))[i1][j]->linearKernel(t, trainData, param);
    else
        fx1 = (*((*model).getModel()))[i1][j]->linearKernel(currentData);

    if (fx1 > maxFx)
    {
        j1 = j;
        maxFx = fx1;
    }
}
fx1 = maxFx;

// calculate i-, j-
i2 = 0;
j2 = 0;
fx2 = 0;
maxFx = -INF;
for (unsigned int i = 0; i < sizeOfyLabels; i++)
{
    if (i == i1)
        continue;

    for (unsigned int j = 0; j < n[i]; j++)
    {
        if ((*param).VERY_SPARSE_DATA)
            fx2 = (*((*model).getModel()))[i][j]->linearKernel(t, trainData, param);
        else
            fx2 = (*((*model).getModel()))[i][j]->linearKernel(currentData);

        if (fx2 > maxFx)
        {
            maxFx = fx2;
            i2 = i;
            j2 = j;
        }
    }
}
fx2 = maxFx;

// downgrade weights each iteration
for (unsigned int i = 0; i < sizeOfyLabels; i++)
    for (unsigned int j = 0; j < n[i]; j++)
        (*((*model).getModel()))[i][j]->downgrade(numIter);

if (1.0 + fx2 - fx1 > 0.0)
{
    // we made a misprediction, push negative class further away, and positive closer!

    if ((*param).VERY_SPARSE_DATA)
    {
        // since we did not create currentData earlier, here we create it to perform updates
        currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
        currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);
    }

    // push the other class further away
    (*((*model).getModel()))[i2][j2]->updateUsingVector(currentData, numIter, -1, param);

    // update the true class weight
    if (fx1 > 0.0)
    {
        // here clone the best weight if the cloning probability allows it
        if ((unsigned int)n[i1] < (*param).BUDGET_SIZE)
        {
            if ((*param).CLONE_PROBABILITY > get_random_probability())
            {
                // clone the winning weight
                budgetedVectorAMM *clonedVector = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                clonedVector->createVectorUsingVector((*((*model).getModel()))[i1][j1]);

                // add the new cloned weight to the model
                (*((*model).getModel()))[i1].push_back(clonedVector);
                n[i1]++;
                clonedVector = NULL;

                // update the clone probability after successful cloning
                (*param).CLONE_PROBABILITY *= (*param).CLONE_PROBABILITY_DECAY;
            }
        }

        (*((*model).getModel()))[i1][j1]->updateUsingVector(currentData, numIter, 1, param);

```

```
                        delete currentData;
                        currentData = NULL;
                    }
                    else
                    {
                        if (n[i1] < (*param).BUDGET_SIZE) // limit number of weights (we found ~20 is a reasonable number per class)
                        {
                            n[i1]++;
                            currentData->updateDegradation(numIter, param);
                            (*((*model).getModel()))[i1].push_back(currentData);

                            countNew++;
                            currentData = NULL;
                        }
                        else
                        {
                            delete currentData;
                            currentData = NULL;
                        }
                    }
                }
                else
                {
                    if (!(*param).VERY_SPARSE_DATA)
                    {
                        // if sparse data then no need for this part, since we didn't even create currentData
                        delete currentData;
                        currentData = NULL;
                    }
                }

                // pruning phase
                if (numIter % (int)(*param).K_PARAM == 0)
                {
                    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);
                    vector <long double> weightNorms, sortedWeightNorms;
                    int numToDelete = 0;

                    // first find the norms of weights
                    for (unsigned int i = 0; i < sizeOfyLabels; i++)
                    {
                        for (vector<budgetedVectorAMM*>::iterator vi = (*((*model).getModel()))[i].begin(); vi != (*((*model).getModel()))[i].end(); vi++)
                        {
                            weightNorms.push_back((double) (*(*vi)).getSqrL2norm());
                        }
                    }

                    // now sort them
                    sortedWeightNorms = weightNorms;
                    sort(sortedWeightNorms.begin(), sortedWeightNorms.end());

                    // find how many before threshold exceeded
                    for (unsigned int i = 0; i < weightNorms.size(); i++)
                    {
                        sumNorms += sortedWeightNorms[i];
                        if (sumNorms > sumThreshold)
                           break;
                        else
                           numToDelete++;
                    }

                    // delete those that should be deleted
                    int counter = 0;
                    bool deleted = false;
                    for (unsigned int i = 0; i < sizeOfyLabels; i++)
                    {
                        for (vector<budgetedVectorAMM*>::iterator vi = (*((*model).getModel()))[i].begin(); vi != (*((*model).getModel()))[i].end();)
                        {
                            long double currNorm = weightNorms[counter++];

                            deleted = false;
                            for (int j = 0; j < numToDelete; j++)
                            {
                                if (currNorm == sortedWeightNorms[j])
                                {
                                    if (n[i] == 1)
                                    {
                                        svmPrintString("Was about to delete all weights of a class, check K_PARAM and C_PARAM parameters!\n");
                                        break;
                                    }
```

```
                                delete (*vi);
                                vi = (*((*model).getModel()))[i].erase(vi);
                                n[i]--;

                                countDel++;
                                deleted = true;
                                break;
                            }
                        }

                        if (!deleted)
                            vi++;
                    }
                }
            }
        }
        timeCalc += clock() - start;

        if (((*param).VERBOSE) && (N > 0))
        {
            sprintf(text, "Number of examples processed: %d\n", numIter);
            svmPrintString(text);
        }
    }

    if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
    {
        sprintf(text, "Epoch %d/%d done.\n", epoch + 1, (*param).NUM_EPOCHS);
        svmPrintString(text);
    }
}
trainData->flushData();

if ((*param).VERBOSE)
{
    sprintf(text, "*** Training completed in %5.3f seconds.\nNumber of weights deleted: %d\n",
(double)timeCalc / (double)CLOCKS_PER_SEC, countDel);
    svmPrintString(text);
    for (unsigned int i = 0; i < sizeOfyLabels; i++)
    {
        sprintf(text, "Number of weights of class %d: %d\n", i + 1, n[i]);
        svmPrintString(text);
    }
}
}
```

6.5.2.4 trainPegasos()

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

Train Pegasos.

Parameters

in	*trainData*	Input training data.
in	*param*	The parameters of the algorithm.
in,out	*model*	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.

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

```
    unsigned long timeCalc = 0, start;
    long double fx, fx1, fx2, maxFx;
    bool stillChunksLeft = true;
    char text[1024];
    budgetedVectorAMM *currentData = NULL;

    // train the model
    for (unsigned int epoch = 0; epoch < (*param).NUM_EPOCHS; epoch++)
    {
        stillChunksLeft = true;
        while (stillChunksLeft)
        {
            stillChunksLeft = trainData->readChunk((*param).CHUNK_SIZE);

            // update the VERY_SPARSE parameter, it is used to speed up the computations of kernels
            //  (of course, in the case of AMM, speeds up linear kernel computation)
            (*param).updateVerySparseDataParameter(trainData->getSparsity());

            // compute observed data dimensionality, where we also account for possible bias term, and check if
            //  we need to expand the current model weights if some new data dimensions were found during loading
            temp = trainData->getDataDimensionality() + (int)(param->BIAS_TERM != 0.0);
            if ((*param).DIMENSION < temp)
            {
                /*sprintf(text, "Extending the model, current: %d\tfound: %d!\n", (*param).DIMENSION, temp);
                svmPrintString(text);*/
                (*model).extendDimensionalityOfModel(temp, param);

                // update the dimensionality
                (*param).DIMENSION = temp;
            }

            N = trainData->N;
            if (numIter == 0)
            {
                // initialize the model
                sizeOfyLabels = (unsigned int) trainData->yLabels.size();
                for (unsigned int i = 0; i < sizeOfyLabels; i++)
                {
                    currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                    vector <budgetedVectorAMM*> perClassWeights;
                    perClassWeights.push_back(currentData);
                    currentData = NULL;
                    (*((*model).getModel())).push_back(perClassWeights);
                }
            }
            else if (sizeOfyLabels != (unsigned int) trainData->yLabels.size())
            {
                // if in the chunks before some class wasn't observed add it here; could happen with small chunks or unbalanced classes
                for (unsigned int i = 0; i < (trainData->yLabels.size() - sizeOfyLabels); i++)
                {
                    currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                    vector <budgetedVectorAMM*> perClassWeights;
                    perClassWeights.push_back(currentData);
                    currentData = NULL;
                    (*((*model).getModel())).push_back(perClassWeights);
                }
                sizeOfyLabels = (unsigned int) trainData->yLabels.size();
            }

            vector <unsigned int> tv(N, 0);
            for (unsigned int ti = 0; ti < N; ti++)
            {
                tv[ti] = ti;
            }

            // randomize the data
            if ((*param).RANDOMIZE)
                random_shuffle(tv.begin(), tv.end());

            start = clock();
            for (unsigned int ot = 0; ot < N; ot++)
            {
                numIter++;
                t = tv[ot];

                i1 = trainData->al[t];
                if ((*param).VERY_SPARSE_DATA)
                {
                    // compute kernels using vectors directly from the budgetedData
                    fx1 = (*((*model).getModel()))[i1][0]->linearKernel(t, trainData, param);
                }
                else
                {
```

```
                    // first create the budgetedVector using the vector from budgetedData, to be used in linearKernel() method below
                    currentData = new budgetedVectorAMM((*param).DIMENSION, (*param).CHUNK_WEIGHT);
                    currentData->budgetedVector::createVectorUsingDataPoint(trainData, t, param);

                    fx1 = (*((*model).getModel()))[i1][0]->linearKernel(currentData);
                }

                //calculate i-, fi-
                fx = 0;
                maxFx = -INF;
                for (unsigned int i = 0; i < sizeOfyLabels; i++)
                {
                    if (i == i1)
                        continue;

                    if ((*param).VERY_SPARSE_DATA)
                        fx = (*((*model).getModel()))[i][0]->linearKernel(t, trainData, param);
                    else
                        fx = (*((*model).getModel()))[i][0]->linearKernel(currentData);

                    if (fx > maxFx)
                    {
                        maxFx = fx;
                        i2 = i;
                    }
                }
                fx2 = maxFx;

                // downgrade the weights
                for (unsigned int i = 0; i < sizeOfyLabels; i++)
                {
                    (*((*model).getModel()))[i][0]->downgrade(numIter);
                }

                // calculate the margin, if misclassified update weights
                if (1.0L + fx2 - fx1 > 0.0L)
                {
                    if ((*param).VERY_SPARSE_DATA)
                    {
                        (*((*model).getModel()))[i2][0]->updateUsingDataPoint(trainData, numIter, t, -1, param);
                        (*((*model).getModel()))[i1][0]->updateUsingDataPoint(trainData, numIter, t, 1, param);
                    }
                    else
                    {
                        (*((*model).getModel()))[i2][0]->updateUsingVector(currentData, numIter, -1, param);
                        (*((*model).getModel()))[i1][0]->updateUsingVector(currentData, numIter, 1, param);
                    }
                }

                if (!(*param).VERY_SPARSE_DATA)
                {
                    // if sparse data then no need for this part, since we didn't even create currentData
                    delete currentData;
                    currentData = NULL;
                }
            }
            timeCalc += clock() - start;

            if (((*param).VERBOSE) && (N > 0))
            {
                sprintf(text, "Number of examples processed: %d\n", numIter);
                svmPrintString(text);
            }
        }

        if ((*param).VERBOSE && ((*param).NUM_EPOCHS > 1))
        {
            sprintf(text, "Epoch %d/%d done.\n", epoch + 1, (*param).NUM_EPOCHS);
            svmPrintString(text);
        }
    }
    trainData->flushData();

    if ((*param).VERBOSE)
    {
        sprintf(text, "*** Training completed in %5.3f seconds.\n", (double)timeCalc / (double)CLOCKS_PER_SEC);
        svmPrintString(text);
    }
}
```

Index