Tutorial to CliqueLib

C++ Library for Ensembles

Aliraja Punjani
Stanislav Peceny
Srinidhi Raghavan

Design using C++
Columbia University

Supervisor: Prof. Bjarne Stroustrup

April 2017

Contents

Chapter 1

Introduction

1.1 Machine Learning

Machine learning studies how to automatically make nearly accurate predictions based on past examples. It considers a set of input data, known as the training set and then tries to predict properties of unknown data based on the inferences made from the training set. The general work-flow of Machine Learning algorithms is given in Fig. 1.1

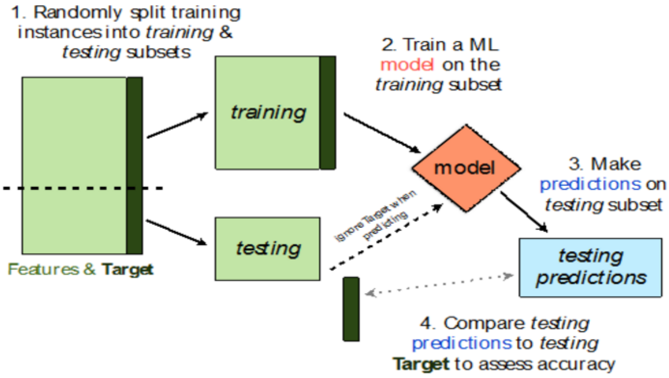

1. Randomly split training instances into *training* & *testing* subsets
Features & Target
training
testing
2. Train a ML model on the *training* subset
model
Ignore Target when predicting
3. Make predictions on *testing* subset
testing predictions
4. Compare *testing* predictions to *testing* Target to assess accuracy

Figure 1.1: ML Process

Machine learning algorithms are further classified into: Supervised and Unsupervised algorithms. In supervised algorithms, every data in the training set has an output value associated with it. A model is prepared through a training process in which it is required to make predictions and is corrected when those predictions are wrong. The training process continues until the model achieves a desired level of accuracy on the training data. If the nature of output is categorical, then it is known as classification, whereas if the output is continuous, it is known as regression. Some examples of regression and classification are as given:

- Classification
    - Handwritten digit recognition: Identify a handwritten digit correctly. Here, the digits are the classes. This is an examples of multi-class classification

- Spam Filtering: Given a email, find if it is spam or not. This is a binary classification. Naive Bayes is most commonly used for spam filtering
- Credit card fraud detection: Given credit card transaction details, find if it is fraudulent or not
- Face detection: Given a image, identify if it contains a face or not
- Cancer Detection: Given the description of tumor, find if it is malignant or benign

- Regression
  - Stock market prediction: Determining the future value of stock based on the past transactions
  - Real estate prediction: Predicting the value of a real estate based on the fluctuations in the market

On the other hand, in unsupervised learning, the data is not labeled and does not have a known result. A model is prepared by deducing structures present in the input data. This may be to extract general rules. It may be through a mathematical process to systematically reduce redundancy, or it may be to organize data by similarity. The most common example of unsupervised learning is image segmentation. Fig 1.2 diagrammatically represents supervised and unsupervised learning.

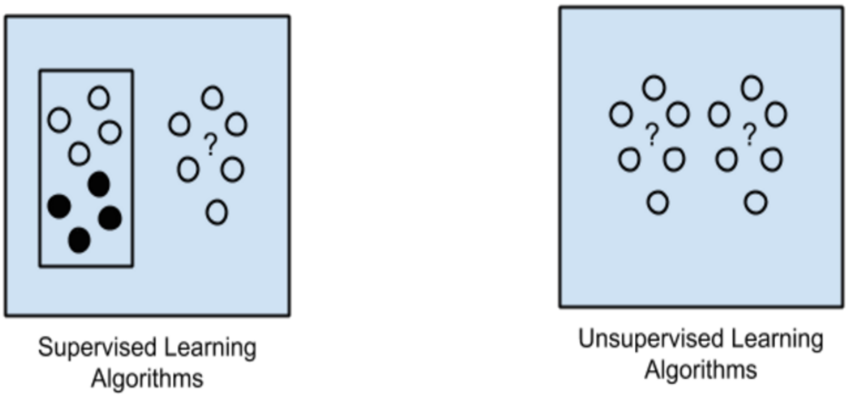

?
Supervised Learning Algorithms
?
?
Unsupervised Learning Algorithms

Figure 1.2: Supervised and Unsupervised Learning

This library explores classification algorithms in depth. In particular, it covers ensemble algorithms, which is a type of classification.

1.2 Classification

Classification is the problem of identifying to which class a new observation belongs, on the basis of training a data-set containing observations whose category membership is known. There are various classification algorithms, some of them are:

1.2.1 KNN

This is the simplest form of classification. It is based on the assumption that if the output of an observation is equal to that of its nearest neighbour. In K-NN, the output is a majority vote of the outputs of the K-nearest neighbors (samples from the training data) of the observations. K-NN does not use any kind of learning, rather it is a simple inference made from the training data. Here, K is a hyper-parameter to be tuned.

1.2.2 Naive Bayes

Naive Bayes classifier uses the Bayesian assumption that the probability of each attribute belonging to a given class value is independent of all other attributes. Despite being a stringent assumption, it works effectively for most applications.

Given a class label y and an n-dimensional observation $X = (x_1, x_2, ..., x_n)$, Bayes theorem states that:

$$P(y|x_1, x_2, ...x_n) = \frac{P(y)P(x_1, x_2, \ldots, x_n|y)}{P(x_1, x_2, \ldots x_n)}$$

Using the naive assumption, we get

$$P(x_i|y, x_1...x_{i-1}, x_{i+1}, ...x_n) = P(x_i|y)$$

Thus,

$$P(y|x_1, x_2, ...x_n) = \frac{P(y)P(x_1|y)P(x_2|y)....P(x_n|y)}{P(x_1, x_2, \ldots x_n)}$$

Since $P(x_1, x_2, \ldots x_n)$ is constant,

$$P(y|x_1, x_2, ...x_n) \propto P(y)P(x_1|y)P(x_2|y)....P(x_n|y)$$

The output label y is the one which maximizes the RHS probability.

Most Naive Bayes assume a Gaussian Distribution as the underlying structure. With this assumption, we have

$$P(x_i|y) = N(\mu, \Sigma)$$

Here, μ and Σ are the parameters

1.2.3 Perceptron

Perceptron is a type of linear classifier (as shown in Fig 1.3). Linear classifiers find a linear decision boundary which tries to separate the data into two classes perfectly. It has to be noted that linear classifiers work perfectly only if the data is linearly separable.

Perceptron algorithm assumes that y = +1, -1. It takes the learning rate 'neta' as a parameter and updates the coefficient weights in batch-wise if any prediction is wrong. The learning rate has to be tuned such that oscillation is avoided but at the same time, the solution is reached fast. The final prediction is made by computing x*weights. If x*weights > 0, the output is labeled 1, else -1

1.2.4 SVM

SVM are classifiers that tries to find the best decision boundary for a given data. The best decision boundary is a boundary which is as wide as possible from the classes. The SVM can have different shapes of boundaries. The shape of the boundary is determined by the kernel. In this library, a linear kernel is used. A linear kernel allows a binary classification. SVM is superior over Perceptron as it allows generalization to as much extent as possible

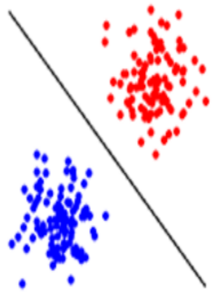


Figure 1.3: Linear Classifiers

1.2.5 Logistic Regression

Logistic Regression is a linear classifier which uses a sigmoid function as the cost function. The functioning of Logistic Regression is just the same as linear regression. The optimum value of the weights is found using gradient descent over the negative of the error.

1.3 Ensemble Methods

Ensemble methods are learning algorithms that classifies using a combination of a set of weak classifiers. Voting, Boosting and Bagging are few types of ensemble methods.

1.3.1 Voting

The most straightforward approach for building an ensemble is training different classifiers. Let us consider that you are given a data set X for classification. Instead of learning a single classifier on this data set, we learn multiple classifiers. For instance, we might train a SVM, perceptron, KNN, decision tree, Neural networks on this data set. During test time, the prediction is done by voting. On a test example, you compute the predictions from all the classifiers. A majority voting is taken amongst all these predictions and is given as the output label. The major advantage of this type of classification is that it is unlikely that many classifiers will do the same mistake.

1.3.2 Bagging

Instead of using different classifiers, bagging trains the same type of classifier on different overlapping subsets of data. It is usually applied on decision trees/ stumps. The basic idea behind bagging is that the bootstrapped datasets will be similar, but not too similar. Hence, it reduces variance, thereby, not overfitting any data.

1.3.3 Boosting

Boosting is the process of taking a bag of weak classifiers and collectively combining them to build a strong classifier. It is more of a framework rather than an algorithm. The most famous boosting algorithm is AdaBoost. It randomly adapts itself to any data we give. It uses a sequence of different weak learners and trains them on modified versions of data. The predictions from all of them are then combined through a weighted majority vote to produce the final prediction.

Chapter 2

Why CliqueLib

Machine Learning aims to automate everything in life. It is used in every other organization to learn interesting features from large datasets. Keeping in mind this enormity, it is very important to have low latency Machine learning provisions. Currently, there are completely developed libraries like scikit-learn in Python, Java-ML in Java. However, due to the overhead involved in these languages, the time efficiency stays less. Due to these reasons, it is advisable to switch over to C++ for Machine Learning applications. C++ already has ML libraries like MLpack and Shark. However, these libraries do not involve customizable ensemble methods. Ensemble methods require a platform as they have wide applications in areas like Face Detection, Spam Filtering.

2.1 Bench-marking with other Libraries

The CliqueLib's KNN and Logistic regression algorithms was compared with those in MLPack and Weka. The observations were as follows:

Library	KNN Run-time (in ms)	Logistic Regression Run-time (in ms)
CliqueLib Armadillo	26149	6541
CliqueLib Vectors	201631	17030
MLPack	27743	Fails
Weka	30000	16206

From the above table it is evident that CliqueLib outperforms most libraries. Besides, CliqueLib was tested on intense applications like Face Detection and Cancer Diagnosis. With these use-cases, it is proven that CliqueLib would be the perfect fit for any real-life application

Chapter 3

Getting Familiarized with CliqueLib

3.1 Importing the Library and Using it

You must first include the corresponding library in your code. The following are the algorithms provided by CliqueLib:

KNN	#include “KNN.h”
Logistic Regression	#include “logistic_regression.h”
SVM	#include “SVM.h”
Naive Bayes	#include “naive_bayes.h”
Perceptron	#include “Perceptron.h”
AdaBoost	#include “AdaBoost.h”
Bagging Classifier	#include “BaggingClassifier.h”
Voting Classifier	#include “VotingClassifier.h”

CliqueLib uses Armadillo for its operations. Armadillo is a high quality optimized linear algebra library in C++ which provides high-end matrix operations in C++ analogous to that in MATLAB. Hence, in order to execute the code you must include the arma namespace. (Note that if you don’t use it, then you will have to append arma:: to every component of the Armadillo library used in the program). This namespace can be included as follows:

```
using namespace arma;
```

3.2 Loading the Dataset

CliqueLib essentially works on CSV files. Hence, you need to process CSV files. In order to load and use Dataset, you must include the preprocessing header

```
#include "Preprocessing.h"
```

The preprocessing component of Cliquelib provides functions for reading CSV files and spliting the data into test and training set.

3.2.1 Reading csv file

Reads the CSV file and converts it into the required format

```
void read_csv(file, X, Y, title, separate_ex, comment)
```

Parameters:

- file - CSV file name in String. The last column of the CSV file must be the labels
- X - The Armadillo Matrix in which you need to store the data
- Y - The Armadillo Column Vector in which you need to store the labels
- title - The Boolean Variable which gives the information if the first row of CSV file is a header or not. The default value of Title is false
- separate_ex - It is a character which denotes the delimiter for the CSV file. It can take the values of ‘,’(comma), ‘;’(semi-colon), ‘ ’(space) and ‘’(newline). The default value of separate_ex is ‘,’(comma)
- comment - The default value of this character is ’’. It indicates that the remainder of line following the character should not be parsed

CliqueLib by default uses few test CSV files in the samples_csv folder. These CSV files can be used for testing. For example:

```
read_csv("samples_csv/Cancer.csv", X, Y, true, ',', '#');
```

3.2.2 Spliting the data into training and testing set

Splits the entire data into training set and test set based on the training set size provided by the user

```
void split_train_test(X, Y, data, trainProportion)
```

Parameters:

- X: This Armadillo matrix is the entire data
- Y: This is the Armadillo column vector denoting the corresponding labels
- data: This is of type Dataset. Dataset consists of the training set (Xtrain, Ytrain) and test set (Xtest, Ytest). These four members are updated by the function
- trainProportion: This double quantity gives the size of the split. It can take any value between 0 to 1

3.3 Defining Classifier and Initializing the parameters

1. KNN: The KNN classifier constructor takes K as a parameter. The default value of K is 1. The object is created as follows:

   ```
   KNN knn_clfr(1);
   ```

2. Logistic Regression: The Logistic Regression constructor takes the learning rate as a parameter. The default value of the learning rate, alpha is 0.01.

   ```
   Logistic logistic_clfr(0.01);
   ```

3. SVM CliqueLib implements a linear SVM using Pegasos Algorithm. Its constructor takes the decay rate, C as the parameter. The default value of C is 0.1

   ```
   SVM svm_clfr(0.1);
   ```

4. Perceptron: The Perceptron has a learning rate as a parameter. The parameter has a default value of 0.01

   ```
   Perceptron perceptron_clfr(0.01);
   ```

5. AdaBoost: The AdaBoost algorithm in CliqueLib uses Decision Stumps as the base classifiers. These base classifiers do not require any external parameter.

   ```
   AdaBoost  adaboost_clfr;
   ```

6. Bagging Classifier: As discussed earlier, bagging uses the sample classifier on multiple samples of overlapping data. It takes the number of classifiers (n_estimators) and the maximum number of samples as parameters (max_samples) respectively. Bagging classifiers are generic types which can be extended to any type of base classifiers. For having a bag of SVM classifiers, you must add the following:

   ```
   BaggingClassifier<SVM>  bagging_clfr(2, 1000);
   ```

7. Voting Classifier: Voting classifier has a collection of Base classifiers as a member. It does not require any parameters in the constructor

   ```
   VotingClassifier  voting_clfr;
   ```

3.4 Training the Dataset

By default all the classifiers in CliqueLib have the same structure for training a dataset.

```
void train(X, Y, epoch)
```

Parameters:

- X - The Armadillo matrix X which represents the data
- Y - The Armadillo column vector Y which represents the labels for the corresponding data. Note that the labels are always either 1 or -1. This is because CliqueLib works in depth for binary classification only
- epoch - Epoch represents the number of iterations taken to get the weight vector. Algorithms like KNN does not require epoch. Hence, the value can be given as 1, which is considered as the default value

3.5 Predicting the labels of the test set

Analogous to the training function, the structure for the predict function is the same for all classifiers

```
void predict(X, Y)
```

Parameters:

- X - The Armadillo matrix X which represents the test data
- Y - The final labels after predicted are stored in this Armadillo column vector. Note again that the labels are either 1 or -1. You simply need to pass an empty column vector while calling the function

Chapter 4

Examples on Real Datasets

In this section, the working of CliqueLib's linear and boosting classifiers are given for large datasets.

4.1 SVM and Perceptron on Linear Dataset

```
#include <iostream>
#include <armadillo>
#include <cmath>
#include <utility>
#include "preprocessing_functions.h"
#include "Dataset.h"
#include "SVM.h"
#include "Perceptron.h"

using namespace arma;

double getAccuracy(colvec& a, colvec& b) {
    return double(accu(a==b))/size(a, 0);
}

int main()
{
    mat X;
    colvec Y;
    read_csv("linear.csv", X, Y, true, ',', '#');

    Dataset data;
    split_train_test(X, Y, data, 0.8);


    cout << "### Running SVM ###" << endl;
    SVM svm_clfr(0.1);
    svm_clfr.train(data.Xtrain, data.Ytrain, 1000);
    colvec preds_svm;
    svm_clfr.predict(data.Xtest, preds_svm);

    auto acc_svm = getAccuracy(data.Ytest, preds_svm);
    cout << "SVM acc:" << acc_svm << endl;



    cout << "### Running Perceptron ###" << endl;
    Perceptron pcptr_clfr;
    pcptr_clfr.train(data.Xtrain, data.Ytrain, 1000);
    colvec preds_pcptr;
    pcptr_clfr.predict(data.Xtest, preds_pcptr);

    auto acc_pcptr = getAccuracy(data.Ytest, preds_pcptr);
    cout << "Perceptron acc:" << acc_pcptr << endl;

    return 0;
}
```

As it was a linearly separable dataset, the accuracy of both SVM and Perceptron was 100

4.2 AdaBoost for Face Detection

In this example, 60% of the dataset is reserved as the training data and the remaining as the test. Adaboost was used with five decision stumps. For a training size of 600, the accuracy obtained was 99.5%.

```
#include <armadillo>
#include <iostream>
#include <vector>
#include <cassert>
#include <cmath>
#include <utility>
#include "preprocessing_functions.h"
#include "Dataset.h"
#include "AdaBoost.h"

using namespace arma;

double getAccuracy(colvec& a, colvec& b)
{
    return (100 * (double(accu(a==b))/size(a, 0)));
}

//use case 1
int main()
{
    mat X;
    colvec Y;
    read_csv("face_detection.csv", X, Y, true, ',', '#');
    X = X.rows(0, 999);
    Y = Y.rows(0, 999);

    Dataset data;
    split_train_test(X, Y, data, 0.6);

    AdaBoost clfr;
    clfr.train(data.Xtrain, data.Ytrain, 5);

    colvec trainPreds;
    clfr.predict(data.Xtrain, trainPreds);
    auto acc = getAccuracy(data.Ytrain, trainPreds);
    cout << "Training accuracy: " << acc << "%" << endl;

    colvec testPreds;
    clfr.predict(data.Xtest, testPreds);
    acc = getAccuracy(data.Ytest, testPreds);
    cout << "Testing accuracy: " << acc << "%" << endl;
    cout << "" << endl;

    return 0;
}
```

Design Using C++

CliqueLib

“*A Machine Learning Library aiming to provide reliable ensemble classifiers*”

Aliraza Punjani (amp2280)
Stanislav Karel Peceny (skp2140)
Srinidhi Srinivasa Raghavan (ss5145)

Compiling the Library

The compilation of the library differs based on the user’s needs. If the user needs to compile all algorithms, and hence ‘make’ all Makefiles present in the library, then he can run the ./compile_cliquelib.sh bash script in the main directory of CliqueLib. If the user would like to compile all algorithms and run all test cases, then he should type ‘make’ in the main folder. The benchmarks can be compiled by running ‘make’ in ./Benchmarking/KNN or ./Benchmarking/LogisticRegression. Should the user like to run the two use cases then he ought to type ‘make’ in either ./UseCases/UseCase1 or ./UseCases/UseCase2. **Please note that the executables must be run in the corresponding folder where the Makefile is present.**

Abstract

CliqueLib is a C++ machine learning library implementing ensemble methods. Ensemble methods use several machine learning algorithms in order to achieve better prediction than could be gained by using any of the constituent algorithms. Hence, the library provides implementation for classification algorithms including KNN, logistic regression, SVMs, and Perceptron classifiers. These algorithms could be either used as standalone algorithms or combined to form an ensemble classifier. C++ libraries like MultiBoost and Mlpack provide support for just Adaboost without allowing the usage of other classifiers for boosting. Our library aims to provide a spectrum of ensembles for the users to choose one that is suitable for their requirements.

1. Problem Statement

Machine learning is currently one of the fastest growing technologies. It has a tremendous number of applications across a diverse range of domains. Python is commonly preferred for Machine Learning applications due to its lesser complexity. However, there are compiled languages like C++ which can be much faster than Python for intricate Machine Learning problems. As of now, there are libraries which provide naive implementations of the Machine Learning suite. However, none of them have a good implementation of ensemble classifiers. In fact, none of the existing C++ libraries implements a basic set of ensemble methods. Ensemble methods essentially enhance the performance of models by strategically combining diverse weak models to solve a computationally intelligent problem like Face Recognition. A generic sturdy library for ensemble methods in C++ can provide for more reliable results and thus lead to the lavish use of CliqueLib in many C++ machine learning applications. CliqueLib's ensemble methods are further unique as they can be easily customized by the user.

2. Design

As C++ is a very powerful and complex language, there are numerous choices that had to be made in relation to programming design. As CliqueLib ought to be used by users in a coherent manner, it is essential that future developers and data scientists can get acquainted with the major decisions involved in the library's design.

2.1. Goals

The goals of CliqueLib can be segmented into four main categories. First, the algorithms had to be efficient, which was one of the main reasons for selecting to use the Armadillo library. Consequently, the API had to be intuitive also for users that are not experts in the field of machine learning or C++. Furthermore, the aim was the quality of algorithms rather than quantity. Ultimately, the user would have at hand customizable ensemble methods that are unique to CliqueLib.

2.2. Benchmarking

As one of the main goals of the library was efficiency, the benchmarking results are presented early in the document. The library used the Armadillo C++ linear algebra library, a high quality tool aiming for ease of use and efficiency, for implementing CliqueLib's algorithms. The efficiency of CliqueLib's library implementation was confirmed by testing the implementations of the KNN and Logistic regression algorithms. In order to obtain a valid comparison, the authors of CliqueLib wrote another implementation of the two algorithms that used vector<vector> > instead of Armadillo's matrices. Furthermore, the performance of the algorithms was further compared with Java's WEKA and C++'s Mlpack machine learning libraries. The Armadillo's implementations worked exceptionally well, and hence represent one of the best decisions made during the library's design process.

All comparisons are made using Kaggle's Cancer Tumor classification dataset having 31 features (columns) and 569 samples (rows).

	KNN Runtime	KNN Accuracy	Logistic Regression Runtime	Logistic Regression Accuracy
CliqueLib (Armadillo)	26.15	*92.93%*	**6.54**	*85.84%*
CliqueLib (vector<vector<>>)	201.63	*92.90%*	17.03	*85.84%*
MLpack (C++)	27.74	*92.92%*	16.20	*85.87%*
Weka (Java)	30.00	*92.92%*	NA	*NA*
Sklearn (Python)	**19.99**	*92.93%*	8.00	*85.87%*

** All times are in milliseconds*

The results of the comparisons are displayed below.

KNN

//==
// Name   : benchmark_knn.txt
// Author   : CliqueLib
// Version   :
// Copyright   :
// Description : Results for benchmarking KNN
//==

The following measurements were taken by running the benchmarking methods on Google Cloud.

The following times are smaller than those presented in the corresponding benchmark_knn.cpp file as the author's computer is extremely slow.

The first two tests compared CliqueLib's Armadillo vs. 'vector<vector> >' implementations.

As you can see, Armadillo greatly increased the performance of the library.

CliqueLib Armadillo:

Runtime: **26149** microseconds

CliqueLib Vectors:

Runtime: **201631** microseconds.

Consequently, the performance was compared with Java's Weka machine learning library and C++'s Mlpack library. Our Armadillo implementation performed very well and even beat Mlpack.

Weka:

Benchmark Weka: KNN

=== Classifier model (full training set) ===

```
IB1 instance-based classifier
using 23 nearest neighbour(s) for classification
```

Runtime: 0.03 Seconds ~ **30000** microseconds

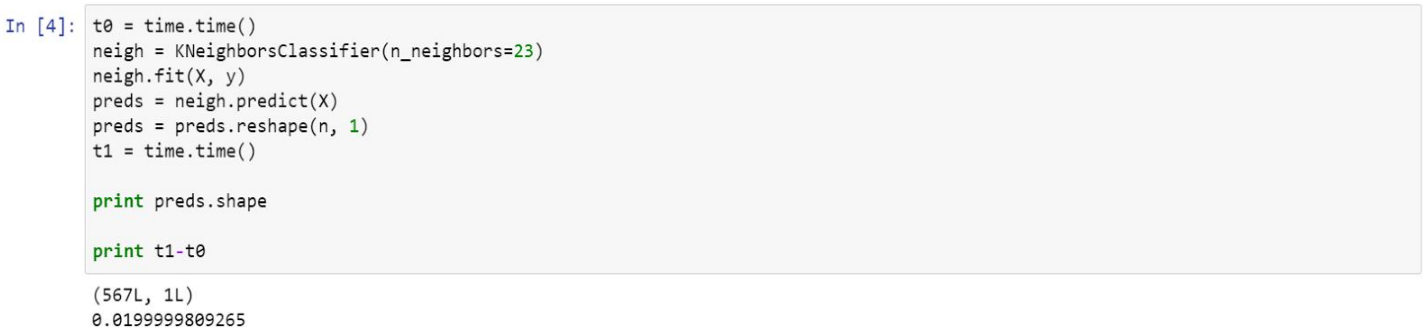
Mlpack:

Runtime: **27743** microseconds

Part of Mlpack's output:

```
[INFO ] Execution parameters:
[INFO ]   algorithm: dual_tree
[INFO ]   distances_file: distances_out.csv
[INFO ]   epsilon: 0
[INFO ]   help: 0
[INFO ]   info:
[INFO ]   input_model_file:
[INFO ]   k: 23
[INFO ]   leaf_size: 20
[INFO ]   naive: 0
[INFO ]   neighbors_file: neighbors_out.csv
[INFO ]   output_model_file:
[INFO ]   query_file:
[INFO ]   random_basis: 0
[INFO ]   reference_file: Cancer.csv
[INFO ]   rho: 0.7
[INFO ]   seed: 0
[INFO ]   single_mode: 0
[INFO ]   tau: 0
[INFO ]   tree_type: kd
[INFO ]   true_distances_file:
[INFO ]   true_neighbors_file:
[INFO ]   verbose: 1
[INFO ]   version: 0
[INFO ] Program timers:
[INFO ]   computing_neighbors: 0.003940s
[INFO ]   loading_data: 0.008077s
[INFO ]   saving_data: 0.014478s
[INFO ]   total_time: 0.027743s
[INFO ]   tree_building: 0.000743s
```

Sklearn KNN:



```
In [4]: t0 = time.time()
        neigh = KNeighborsClassifier(n_neighbors=23)
        neigh.fit(X, y)
        preds = neigh.predict(X)
        preds = preds.reshape(n, 1)
        t1 = time.time()

        print preds.shape

        print t1-t0

        (567L, 1L)
        0.0199999809265
```

Logistic Regression

```
//======================================================================
// Name        : benchmark_logisticregression.txt
// Author      : CliqueLib
// Version     :
// Copyright   :
// Description : Results for benchmarking Logistic Regression
//======================================================================
```

The following measurements were taken by running the benchmarking methods on Google Cloud.

The following times are smaller than those presented in the corresponding benchmark_logisticregression.cpp file as the author's computer is extremely slow.

The first two tests compared CliqueLib's Armadillo vs. 'vector<vector> >' implementations.

As you can see, Armadillo greatly increased the performance of the library.

CliqueLib Armadillo:

Runtime: **6541** microseconds

CliqueLib Vectors:

Runtime: **17030** microseconds.

Consequently, the performance was compared with Java's Weka machine learning library and C++'s Mlpack library. Our Armadillo implementation performed very well and was on par with Mlpack.

Weka:

Fails

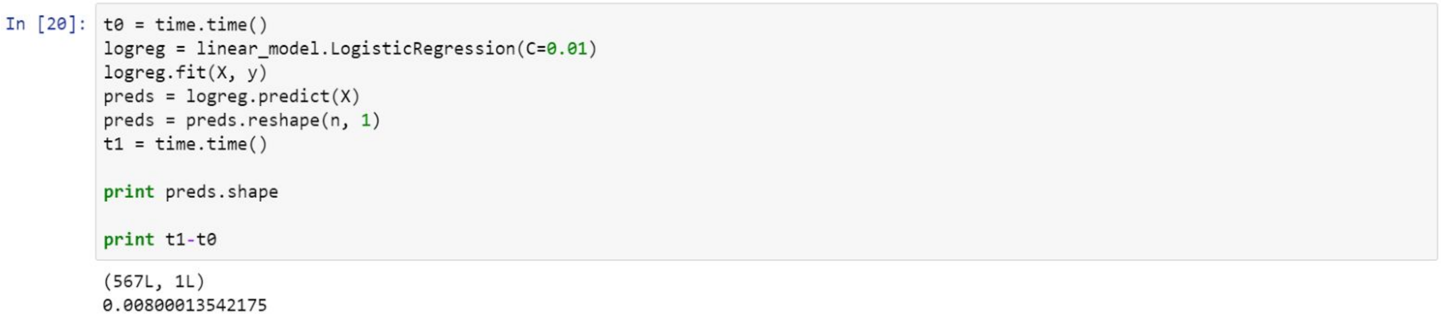
Mlpack:

Runtime: **16206** microseconds

Part of Mlpack's output:

```
[INFO ] Execution parameters:
[INFO ]   batch_size: 50
[INFO ]   decision_boundary: 0.5
[INFO ]   help: 0
[INFO ]   info:
[INFO ]   input_model_file:
[INFO ]   labels_file:
[INFO ]   lambda: 0
[INFO ]   max_iterations: 100
[INFO ]   optimizer: sgd
[INFO ]   output_file: output.txt
[INFO ]   output_model_file: model.txt
[INFO ]   output_probabilities_file:
[INFO ]   step_size: 0.01
[INFO ]   test_file: Cancer.csv
[INFO ]   tolerance: 1e-10
[INFO ]   training_file: Cancer.csv
[INFO ]   verbose: 1
[INFO ]   version: 0
[INFO ] Program timers:
[INFO ]   loading_data: 0.015230s
[INFO ]   logistic_regression_optimization: 0.000266s
[INFO ]   saving_data: 0.000103s
[INFO ]   total_time: 0.016206s
```

Sklearn Logistic Regression:



```
In [20]: t0 = time.time()
         logreg = linear_model.LogisticRegression(C=0.01)
         logreg.fit(X, y)
         preds = logreg.predict(X)
         preds = preds.reshape(n, 1)
         t1 = time.time()

         print preds.shape

         print t1-t0

         (567L, 1L)
         0.00800013542175
```

In sum, CliqueLib’s Armadillo algorithm implementations match the performance of other well-established libraries.

2.3. Concepts

CliqueLib uses concepts to constrain what base classifiers could be used with the Bagging meta-estimator. Library defines the concept of a “Classifier” which enforces that template argument to Bagging Classifier provide train and predict functions in a certain expected format. This makes sure that compilation errors are terse and comprehensible when the use attempts to construct a Bagging meta-estimator with an improper base classifier.

2.4. SmartPointer

Handles were used to represent ownership wherever possible. shared_ptrs were used where pointers were unavoidable, to make sure the library does not leak resources.

2.5. Interface

To get data out of functions, the library defines most interfaces to accept references to empty containers (in most cases armadillo column vectors arma::colvec for labels) and expects the functions to fill them up with relevant data. This approach was selected as there were no guarantees of move semantics being implemented for armadillo matrices and vectors. Similarly, to pass huge datasets into the function, const references were employed.

A Dataset struct is defined to get train and test splits cheaply and tersely out of the preprocessing functions.

2.6. Armadillo

Armadillo is a high quality linear algebra library for the C++ language. The library aims for a balance between efficiency and ease of use. The library has syntax similar to Matlab and is particularly suitable for algorithm development in C++. Hence, the library is ideal for use in writing machine learning algorithms also due to its efficient classes for vectors and matrices.

The algorithms implemented in Armadillo greatly improved the efficiency of the library as demonstrated in the Benchmarking section and also simplified the algorithm coding process. As CliqueLib library had algorithm implementations both in Armadillo matrices as well as in vector<vector> >, the major finding was that Armadillo is an indispensable tool in the field of machine learning.

2.7. Vector Implementations

For benchmarking purposes, some algorithms were written not only in Armadillo matrices, but also in vector<vector> >. These implementations were not ideal for machine learning as the datasets need to be processed in a manner that is well-suited by Armadillo's implementation.

2.8. Vectorization

Vectorization is a technique of generalizing operations on scalars to apply transparently to vectors, matrices, and higher-dimensional arrays. The core idea behind vectorization is that operations apply at once to an entire set of values. This makes it a high-level programming model as it allows the programmer to think and operate on whole aggregates of data, without having to resort to explicit loops of individual scalar operations. It looks to group data and apply a uniform handling to exploit the properties of data where individual elements are similar or adjacent.

Below are the snippets from the loop based and vectorized implementation of KNN.

Loop based KNN:

```
for (unsigned int i = 0; i < k_neighbors.size(); i++)
 {
      float neighbor_label = k_neighbors[i][k_neighbors[i].size() - 1];
      if (classification.find(neighbor_label) != classification.end())
         classification[neighbor_label] +=1;
      else
         classification[neighbor_label] = 1;
 }

   //set to minimum
   int largest_val = numeric_limits<int>::min();
   float largest_class = numeric_limits<float>::min();
```

```
for (unsigned int i = 0; i < k_neighbors.size(); i++)
{
    if (classification[k_neighbors[i][k_neighbors[i].size() - 1]] > largest_val)
    {
        largest_val = classification[k_neighbors[i][k_neighbors[i].size() - 1]];
        largest_class = k_neighbors[i][k_neighbors[i].size() - 1];
    }
}
```

Vectorized KNN:

```
mat X2 = sum(Xtrain % Xtrain, 1);
        mat Y2 = (sum(Xtest % Xtest, 1)).t();
        mat XY = 2* Xtrain * Xtest.t();
        mat dist = XY.each_col() - X2;
      dist = dist*(-1);
        dist = dist.each_row() + Y2;
```

Vectorized code is not only terse but can be many folds faster than it’s equivalent loop-based implementations as exhibited in the bench markings.

2.9. Algorithms

The library is aimed to provide customizable classification ensembles. Thus, all the prominent classification algorithm were implemented. AdaBoost was implemented as an out of the box ensemble for users who weren't interested in customizing ensembles but rather just use a state of the art ensemble for their application. It was made sure that the library had decent diversity in terms of the type of the algorithm that it provides. Library was constrained to binary classifiers to make sure all algorithms would be compatible with the meta estimator. An alternate implementation using vectors was included primarily for the purpose of benchmarking and comparison.

2.9.1. KNN

K-Nearest Neighbors (KNN) is a simple and robust classifier that is frequently used as a benchmark for more complex classifiers such as Support Vector Machines (SVM) or Artificial Neural Networks. Despite its simplicity, KNN outperforms many classifiers and is used in a wide range of applications such as economic forecasting or data compression. Due to its efficiency and wide range of applications, KNN is incorporated as one of CliqueLib's classifiers.

In the classification setting, the KNN algorithm essentially boils down to forming a majority vote between the K most similar instances to a given "unseen" observation. Similarity is defined using distance metrics like Euclidean distance. Formally, given a positive integer K and an unseen observation x, KNN classifier performs the following steps:

- It goes through the entire dataset and finds the distance between x and every other point
- The set of K closest points to x is taken into consideration
- It finds the labels associated with each of these K points
- A majority voting is done among these labels. The label with the maximum vote is chosen as the predicted output

The major design challenge for implementing KNN is the computational complexity involved in every unseen observation. Consider predicting the label of one unseen instance. It takes O(n) time for finding the distance of x from every point in the training data and O(n log k) to get the lowest k distances. Thus, the total time taken for one unseen observation would be O(n log k). If there are m unseen observations, then the total time taken is O(mn log k). If the size of the data is large, then this means that KNN would take several minutes to predict the label. We addressed this challenge by using vectorized implementation of distance computation.

Let Xtrain be n * d matrix of the training data, Ytrain be the n * 1 matrix of training labels and Xtest be the m * d matrix of the test data. CliqueLib uses the square of Euclidean distance as the similarity measure of KNN. This can be computed by using the expansion of the algebra formula (a-b)^2. This can be computed as follows:

1. Find X2 = Xtrain*Xtrain’ (n*n matrix)
2. Find Y2 = Xtest*Xtest’   (m*m matrix)
3. Find XY = Xtrain*Xtest’ (n*m matrix)
4. Return X2 - 2*XY + Y2 after doing dimensionality matching

This vectorized implementation of KNN improves the performance multi-folds.

2.9.2. Logistic Regression

Logistic Regression is a regression model where the decision boundary is linear and the outputs are categorical. It uses gradient descent on every data to find the best possible value of the weight vectors which fits the given model. However, apply gradient descent for every observation in the training data takes a long time because of the number of loops involved. In order to get rid of these loops, a vectorized implementation is used. This was done as follows:

1. Find the transpose of Xtrain. This transpose is a d * n matrix
2. Find Xtrain * w which is a n * 1 column vector
3. Find sigmoid (Xtrain*w) which applies the sigmoid function on every entry of the vector
4. Find the difference between Y and sigmoid(Xtrain*w)
5. Add transpose(Xtrain) * (Y - sigmoid(Xtrain*w)) to the weight vector

2.9.3. Perceptron

Perceptron is a linear classifier algorithm with the labels {-1, 1}. It allows for online learning, i.e. it enables to process one element at a time from the training set. As a result, two loops would be used to implement the perceptron algorithm. There is no way to optimize this looping. There is another variant of perceptron known as the batch algorithm, which allows a simultaneous update of many observations. However, batch algorithms are not effective in most of the cases, and hence it was decided to move ahead with the online algorithm itself. Despite the use of the two for loops, the classifier trains in a time comparable to that of other classifiers.

2.9.4. SVM

SVM is a powerful linear classifier with enormous advantages. It uses the kernel trick to find the best possible decision boundary and also avoids overfitting by using a regularization parameter. It is very helpful in text categorization and image classification. CliqueLib implements SVM using the linear kernel. However, the computational overhead involved with fitting the data is huge. In order to improvise it, the Pegasos algorithm is used to implement SVM. Pegasos algorithm is a linear algorithm which runs iteratively in a fashion similar to that of Perceptron, until the algorithm converges. With the Armadillo library, the entire implementation of SVM was vectorized.

2.10. Ensembles

2.10.1. AdaBoost

AdaBoost fits a sequence of weak classifiers on a sequence of modified versions of data. The predictions from all the data are combined to form a weighted majority voting, which eventually produces the final prediction. Typically, AdaBoost uses decision stumps as the weak classifiers. A decision stump is essentially a one-level decision tree. These stumps were created as a separate class for AdaBoost. Decision stumps make predictions based on just one input feature. The input feature to the decision stump is stored in the variable dimension. The error and threshold for each classifier/stump are stored as well. The AdaBoost considers a collection of these stumps as the Base Weak Classifier. To maintain the structure of the train function, it is assumed that in every epoch, one base weak classifier is learned. Thus, in each iteration/epoch, the AdaBoost builds a stump on the data and the modified weights. AdaBoost is used for large scale applications like Spam Filtering and Face Detection. Hence, its implementation was vectorized to reduce the latency to as much extent as possible. The run-time complexity for this AdaBoost implementation is O(n*t) where n is the size of the input data and t is the number of iterations/ number of classifiers. This complexity must not be confused with the quadratic one as t << n.

2.10.2. Bagging Classifier

The Bagging classifier or more formally, the Bagging meta-estimator was given a templated implementation to allow for multiple base classifiers while taking advantage of “Concepts”. The template argument is required to satisfy the constraints set by the “Classifier” concept as described in the section for Concepts.

The implementation for the Bagging classifier was designed to be simple yet flexible enough to act as a meta-estimator over any classifier that could fulfill the simple constraints laid down by the Classifier concept. The data members for the class were kept to minimum, just the sequence of base classifiers which were variation of models on the same base classifier fit/trained on a random subsample of the training data.

Predicting would involve iteratively predicting on each of the saved models and averaging/taking majority vote on these predictions.

2.10.3. Voting Classifier

Voting classifiers combine conceptually different machine learning classifiers and use a majority vote to predict the class labels. Such a classifier can be useful for a set of equally well performing model in order to balance out their individual weaknesses.

The requirements of a voting classifier are as follows:

- Ability to add any type of classifier at any given point of time
- Flexibility to perform both majority voting and averaging
- Possibility to train all the classifiers under similar conditions

CliqueLib incorporates the first feature by having an interface to add the classifier. This is done by using the function: add_classifier. A pointer to the base classifier is sent as the argument to the add_classifier. As it is enforced to have just labels +1 and -1, the concept of averaging and majority voting mean one and the same. All the classifiers have the same structure for predict and train functions and hence, are always trained under similar conditions.

3. Use Cases

The two use cases can be found in the UseCases folder of CliqueLib's library. The use cases can be compiled either by running the ./compile_cliquelib.sh script in the library's root folder or by running 'make' in ./UseCases/UseCase1 or ./UseCases/UseCase2. The use cases are described below.

3.1. Use Case 1

This use case utilizes the famous Viola Jones Face Detection technique, which uses AdaBoost for learning to identify a human face. A sample of 1000 images were taken and formatted into a CSV file using MATLAB. This CSV file was given as the input to the use case application. The data was further divided into 60% training data and 40% test data. The AdaBoost was trained with five iterations to predict if a given image consisted of a face or not. The accuracy of AdaBoost on this data was observed to be 99.5%.

3.2. Use Case 2

The second use case drew 569 samples from Cancer.csv which has 31 cancer tumor features like parameter, texture, or compactness. The aim was to test CliqueLib's algorithms as well as ensembles and classify the tumor as Malignant (-1) or Benign (1). Hence, this was a binary classification example, which started by loading the dataset and splitting it into training and testing sets with ratio 80%-20%.

Each classifier or ensemble was tested for accuracy and the results are displayed below. The performance of the algorithms was rather satisfactory.

AdaBoost was selected with 5 decision stumps as empirically this was the number that worked well for this particular dataset. As a result, AdaBoost yielded the best results from all available methods.

```
Training AdaBoost with 5 decision Stumps...
Testing accuracy: 97.3451%
```

For KNN, the parameter K was selected to be 23 as this number represents the square root of the dataset size, an optimal selection for KNN. KNN's performance was relatively satisfactory, as well.

```
Training KNN with K=23 ...
Testing accuracy: 92.9204%
```

Larger number of epochs generally enables the algorithms to reach a more reliable model as the coefficients converge with more iterations. The learning rates or regularization parameters of the following algorithms are relatively standard and are small enough to allow the coefficients reach the desirable value.

```
Training SVM with regularization parameter=0.1 and 1000 epochs...
Testing accuracy: 85.8407%
```

```
Training Perceptron with learning rate 0.01 and 1000 epochs...
Testing accuracy: 92.9204%
```

```
Training Logistic regression with learning rate 0.01 and 1000 epochs...
Testing accuracy: 85.8407%
```

The remaining methods focus on the main aim of the library to enable the user to customize various ensemble classifiers. The Bagging Classifier over 5 KNN models performed particularly well.

```
Training Bagging Classifier over 5 KNN models and 1000 epochs...
Testing accuracy: 94.6903%
```

```
Training Bagging Classifier over 5 SVM models and 1000 epochs...
Testing accuracy: 88.4956%
```

```
Training Voting Classifier over KNN, Perceptron and Logistic...
Testing accuracy: 92.0354%
```

In sum, the Bagging Classifier over 5 KNNs and AdaBoost yielded the best results.

4. Unit Tests

The library has been tested diligently as each feature was included in the library code. The tests for all algorithms and functions are present in the ./tests folder. The tests are compiled with ‘make’ in the library’s root directory. Consequently, the tests are run with ./main. The tests can be further compiled by running the script ./compile_cliquelib.sh, which is further going to compile all use cases and benchmarking methods, which have a separate Makefile.

The test files are called by the main function in main.cpp in the library’s root directory. Each test file further includes tests.h, which contains the includes of all the library’s algorithms.

The tests are quite comprehensive. Nevertheless, a more formal structure is planned for the library which will utilize Catch, a modern framework for unit tests, TDD, and BDD. The corresponding link is below:

https://github.com/philsquared/Catch

5. Documentation

The Doxygen documentation is placed in the Documentation folder in the root directory of the CliqueLib library, submitted in the .zip folder. The documentation can be also viewed in the same directory on our GitHub repository:

https://github.com/skp2140/CliqueLib

6. Group Review

Stan

I greatly enjoyed working with the group. My primary contributions consisted of setting up the structure of the library including Makefiles (benchmarking, use cases, script, library's compilation), file structure, and the library environment, implementations of KNN, Logistic Regression, and Naive Bayes (eventually removed) vector algorithms, benchmarking with armadillo and vector implementations alongside Weka and Mlpack libraries, testing suite, refactoring the codebase, Doxygen documentation, and design document. I also added the preprocessing functions (reading csv and the initial version of splitting test and training function).

Aliraza

Contributions from my end involved implementing the AdaBoost suit decision stumps and putting them together using the AdaBoost algorithm as described by Yoav Freund, et al here. Also implemented the customizable Voting and Bagging Classifiers as well as the Concept of a "Classifier" as used by the Bagging classifier implementation. In terms of the preprocessing functionality, I was responsible for the split_train_test function for splitting and random shuffling the data set.

Also implemented the two use cases for the library, face detection with AdaBoost and Cancer tumor classification with all the algorithms. This involved finalizing and curating the data sets for our implementation and trial and tests to select the most optimal hyper parameters for the algorithms.

Finally, I contributed to the design documentation and project report.

Srinidhi

I worked on the implementations of the classifiers - SVM, KNN, Perceptron and Logistic Regression using the Armadillo library. Besides, implementing and testing these classifiers, I wrote the tutorial and contributed to the documentation.

7. Future work

Future work on CliqueLib involves:

- Extending the library to multi-class classification
- Adding a regression module along with classification
- Further comprehensive testing with Catch as described in the section on unit tests

8. References

"Armadillo: C++ Linear Algebra Library". [Web.] arma.sourceforge.net, Accessed April 27, 2017.

AdaBoost examples taken from slides by Prof. Daniel Hsu, CS Dept., Columbia University

Logistic Regression implementation as per ML course by Prof. Andrew NG, Stanford University

Freund, Yoav; Schapire, Robert E (1997). "A decision-theoretic generalization of on-line learning and an application to boosting". Journal of Computer and System Sciences. 55: 119

Ensemble explanations paraphrased from "A Course in Machine Learning", Hal Daumé III

"Machine Learning Mastery". [Web.] http://machinelearningmastery.com/, Accessed April 27, 2017.

"Mlpack: A Scalable C++ Machine Learning Library". [Web.] http://www.mlpack.org/, Accessed April 27, 2017.

"Weka - University of Waikato". [Web.] http://www.cs.waikato.ac.nz/ml/weka/, Accessed April 27, 2017.

API design for machine learning software: experiences from the scikit-learn project, Buitinck et al., 2013

"KNN - A complete guide to KNN applications in python and R". [Web.] https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/ , Accessed April 27, 2017.

9. Commit History

The commit History can be viewed on the following link:

https://github.com/skp2140/CliqueLib/commits/master

The repository is public.