APC 524 User Manual: The Automatic Statistical Learner

Bill Eggert, Ariel Gewirtz, Nina Gnedin, Chase Perlen,
Noemi Vergopolan Rocha, and Andreas Winther Rousing

January 16th, 2017

This manual is a brief introduction to the installation and use of the automatic statistical learner. In the following sections you will find instructions to compile and execute the software, details about the prepossessing modules and the model fitting modules.

1 Instructions to User

This software was developed to be used in command line environment, and here you will find instructions on how to compile, execute, and modify initial configuration options according to the desired simulations.

1.1 Compile C++ Modules

make

1.2 Edit Configuration File

The figure below shows an example of the configuration file, which could be acesses by:
vim config/config.json

It's possible to customizing your initial configurations by:

- Set your training and testing directories in *traning_dir* and *testing_dir* options
- Set your image file and label files in *train_img* and *train_lbl* oprtions for the training dataset and *test_img* and *test_lbl* options for the testing dataset

1.3 Execute

There is two ways you can execute the file. The first is through a python driver, which will read input data directories from confi.json:
python ./driver.py

Alternatively, you can execute and pass input file folders by hand using:
./main train_dir test_dir train_lbl train_img test_lbl test_img

The user will be able to set additional configurations parameters through i/o communication with the user via terminal. The user will select yes (0) or no (1) for the following options:

- Processing Modules: No Processing, Histogramming, and Gaussian Smoothing.
- Model Fitting: Linear Regression, Regularized Regression, Logistic Regression, and kNN.

The user will be asked to input:

- *How many iterations do you want to run gradient descent?*
- *How large do you want the batches to be in the gradient descent algorithm?*
- *What is the threshold for convergence of gradient descent? (eg 10e-3)*
- *What do you want the initial learning rate for gradient descent? (eg 10e-2)*
- *How many folds would you like to use for cross validation? (eg 4)*

2 Data Processing

This section presents details on the data processing modules:

- **data_process_base.h:**

  Virtual class for all derived data processing types.

  - **Variables:** Train data and labels, test data and labels, flag for train having been processed, flag for test having been processed
  - **Functions:** Accesssors and modifiers for about variables (not including flags)
  - **Process function:** It has no inputs but runs off of internal flags to avoid user interaction (i.e. user processing wrong data or data twice). Running once processes train, the second time processes test, all other runs after that do nothing. The exact mechanics of the function depend on the processing type. Processed data overwrites the data members - *no_processing* will always keep the raw data.

- **no_processing.h/.cpp:**

  The process function here returns the raw data.

- **gaussian_smoothing.h/.cpp:**

  Uses seperability of Gaussian kernel to do horizontal/vertical convolutions seperately. Implements a sliding window approach to avoid needing additional data storage. Kernel is 5x5. Mirroring is used for border conditions. Process function does not change image or dataset dimensions - just the pixel values contained therein.

- **histogram.h/.cpp:**

  Frequency counts of pixels in each image. Resulting histogram is matrix of *n* image X *k* possible pixel values (0-255 since we are using grayscale). The histogram is added to the front of the dataset being processed - thereby adding one image to the vector length. This is changed in the driver function for compatibility with the modeling portion.

3 Model Fitting

3.1 Models

This section presents details on the data the model fitting modules:

- **model.h:**

  It's a pure virtual class for Model objects.

  - **Variables:** -
  - **Functions: Destructor:**
    - *predict:* It will take a matrix of examples and return a vector of predicted labels for them
    - *set_Params:* It will set the kth entry in the parameter vector to the vector p
    - *get_Params:* It will take nothing and return the vector of vectors where the parameters for the model are stored
    - *get_num_examples:* It will take nothing and return the number of training examples provided to the model
    - *getTrainset:* will take nothing and return the matrix of training examples
    - *getLabels:* will take nothing and return the vector of training labels.
    - *getLabelSet:* will take nothing and return the set of all possible labels.
  - **GradientModel.h:**

    It's a pure virtual class for GradientModel objects. It inherits from Model.h with the following (notable) exception

    - **Functions:**

    * *gradient* to takes two ints, the lower and upper indices of examples to use in computing the gradient

- **KNN.h/.cpp:**
  - **Variables:**
    * *trained:* Flag to set whether the model has been trained yet, to make sure that normalization applies to the testing set the same as the training set.
    * *normalize:* Flag to set whether the data should be normalized (a flag set in the main portion of the program)
    * *optim:* the optimizer object associated with the model.
    * *num_examples:* the number of training examples
    * *x:* the matrix of training data
    * *y:* the vector of training classification labels:
    * *params:* For KNN, this is just the k chosen by the optimizer used to choose k nearest neighbors.
    * *label_set:* a set of all possible labels for classification
    * *tr_means:* row vector of means from the training examples
    * *tr_stdev:* row vector of standard deviations from the training examples
    * *remove:* vector of booleans representing whether columns in x are collinear or uninformative and should be removed
    * *num_regressors:* number of informative columns, used to allocate matrix size
    * *initial_regressors:* number of total initial columns (features) before trimming
  - **Functions: Constructor:**
    It takes in training examples, training labels, an optimizer object, and a flag saying whether the data should be normalized. It sets several member variables, normalizes the data if the flag is true, shuffles the examples in case the user has provided them in order, and calls the fit method. Doesn't return anything.
    * *Fit:* takes no arguments, calls the Optimizer's method fitParams, passing itself as an object.
    * *Predict:* takes a matrix of test examples as an argument. If the normalize flag is true, it normalizes the testing data. It calculates the distance matrix for each test example to each train example. It calls internal_predict and returns the output from that function. Predict is called from the main program.
    * *InternalPredict:* takes a matrix of test examples, a matrix of training examples, k, a vector of the labels for the training examples, and a distance matrix of the L2 norm between each testing and training example. It finds the mode of the labels of the k closest neighbors to each test example from the training set and sets that as the predicted label for that test example. This returns a vector of the predicted labels for the test examples. This is a private function.

    * *Predict_on_subset:* Takes a matrix of test samples, a matrix of training examples, k, a vector of the labels for the training examples, and a distance matrix of the L2 norm between each testing and training example. It passes these arguments straight into internal_predict and returns the output from that function. Intended to be called from the CrossValidation Optimizer object– ie predict labels on a subset of the training data.
    * *set_Params:* a public method to set k, called by the Optimizer object to set the best k it found as a member variable for the KNN object. It takes an integer k , which is NOT the k to set–this is just the index in the parameter vector to set equal to the second parameter– a vector which here is just a vector containing the best k for KNN, as found by CrossValidation. This doesn't return anything.
    * *get_Params:* takes nothing, returns the vector of parameters. The first element here is a vector, the first element is the k selected by cross validation.
    * *getTrainset:* public method which takes no parameters and returns x.
    * *getLabels:* public method which takes no parameters and returns y.
    * *get_num_examples:* public method which takes no parameters and returns num_examples
    * *getLabelSet:* public method which takes no parameters and returns label_set.
    * *standardize:* private method to standardize a matrix of examples.
  - **LinearRegression.h/.cpp:**
    * **Variables:**
      · *trained:* Flag to set whether the model has been trained yet, to make sure that normalization applies to the testing set the same as the training set.
      · *normalize:* Flag to set whether the data should be normalized (a flag set in the main portion of the program)
      · *optim:* the optimizer object associated with the model.
      · *num_examples:* the number of training examples
      · *x:* the matrix of training data
      · *y:* the vector of training classification labels:
      · *params:* For LinearRegression, this is the β parameter that multiplies all the pixel values of each example in making a prediction
      · *label_set:* a set of all possible labels for classification
      · *tr_means:* row vector of means from the training examples
      · *tr_stdev:* row vector of standard deviations from the training examples
      · *remove:* vector of booleans representing whether columns in x are collinear or uninformative and should be removed
      · *remove:* a boolean vector indicating whether the column is constant and hence is removed and should be removed when making predictions.
      · *num_regressors:* number of informative columns, used to allocate matrix size

        - *initial_regressors:* number of total initial columns (features) before trimming
        - *l1:* the lasso penalty
        - *l2:* the ridge penalty
    - **Functions: Constructor:**
      It takes in training examples, training labels, an optimizer object, the l1 and l2 penalties, and a flag saying whether the data should be normalized. It sets several member variables, normalizes the data if the flag is true, shuffles the examples in case the user has provided them in order, and calls the fit method. Doesn't return anything.
        - *get_exactParams:* if the fitted model is ridge or standard linear regression, returns the closed form solution for the model
        - *sign_fct:* auxillary function for computing the gradient, returns if its input is > 0 (+1), < 0 (-1), or == 0 (0)
        - *standardize:* standardizes the data so each column has mean 0 stdev 1 and removes extraneous columns. stores tr_means and tr_stdev for normalizing the testing data in future
        - *Fit:* takes no arguments, calls the Optimizer's method fitParams, passing itself as an object.
        - *Predict:* takes a matrix of test examples as an argument. If the normalize flag is true, it normalizes the testing data. It calculates the distance matrix for each test example to each train example. It calls internal_predict and returns the output from that function. Predict is called from the main program.
        - *set_Params:* a public method to set the parameter of class k, in this case always 0, called by the Optimizer object to set the result of the current iteration of gradient descent.
        - *get_Params:* takes nothing, returns the vector of parameters. The first element here is a vector, the first element is the k selected by cross validation.
        - *getTrainset:* public method which takes no parameters and returns x.
        - *getLabels:* public method which takes no parameters and returns y.
        - *get_num_examples:* public method which takes no parameters and returns num_examples
        - *getLabelSet:* public method which takes no parameters and returns label_set.

- **LogisticRegression.h/.cpp:**
  - **Variables:**
    - *trained:* Flag to set whether the model has been trained yet, to make sure that normalization applies to the testing set the same as the training set.
    - *normalize:* Flag to set whether the data should be normalized (a flag set in the main portion of the program)

  * *optim:* the optimizer object associated with the model.
  * *num_examples:* the number of training examples
  * *x:* the matrix of training data
  * *y:* the vector of training classification labels:
  * *params:* For LogisticRegression, this is a vector of β parameters fit by class. Each class β is the multiplier that maximizes the one-v-rest odds $1.0/1.0 + exp(-\beta X)$.
  * *label_set:* a set of all possible labels for classification
  * *tr_means:* row vector of means from the training examples
  * *tr_stdev:* row vector of standard deviations from the training examples
  * *remove:* vector of booleans representing whether columns in x are collinear or uninformative and should be removed
  * *remove:* a boolean vector indicating whether the column is constant and hence is removed and should be removed when making predictions.
  * *num_regressors:* number of informative columns, used to allocate matrix size
  * *initial_regressors:* number of total initial columns (features) before trimming

- **Functions: Constructor:**
  It takes in training examples, training labels, an optimizer object,and a flag saying whether the data should be normalized. It sets several member variables, normalizes the data if the flag is true, shuffles the examples in case the user has provided them in order, and calls the fit method. Doesn't return anything.
  * *get_exactParams:* if the fitted model is ridge or standard linear regression, returns the closed form solution for the model
  * *ovr_labels:* auxillary function for storing labels for gradient descent. For taking the gradient of class i, all examples from class 1 have ovr_label 1 and the rest 0
  * *standardize:* standardizes the data so each column has mean 0 stdev 1 and removes extraneous columns. stores tr_means and tr_stdev for normalizing the testing data in future
  * *Fit:* takes no arguments, calls the Optimizer's method fitParams, passing itself as an object.
  * *Predict:* takes a matrix of test examples as an argument. If the normalize flag is true, it normalizes the testing data. It calculates the distance matrix for each test example to each train example. It calls internal_predict and returns the output from that function. Predict is called from the main program.
  * *set_Params:* a public method to set the parameter of class k, called by the Optimizer object to set the result of the current iteration of gradient descent.
  * *get_Params:* takes nothing, returns the vector of parameters. The first element here is a vector, the first element is the k selected by cross validation.
  * *getTrainset:* public method which takes no parameters and returns x.

    * *getLabels:* public method which takes no parameters and returns y.
    * *get_num_examples:* public method which takes no parameters and returns num_examples
    * *getLabelSet:* public method which takes no parameters and returns label_set.

3.2 Optimizers

This section presents details on the optimizers modules:

- **Optimizer.h:**
  Abstract class for Optimizer objects.
  - **Variables:** None
  - **Functions:** Destructor
    * *fitParams:* takes a model object. In the specific implementations of Optimizer objects this function fits the parameters in the model that it is associated to.
- **CrossValidation.h/.cpp:**
  Conducts cross validation to select the best hyperparameter from a given interval.
  - **Variables:**
    * *param_range_start:* the start of the possible values for the hyperparameter
    * *param_range_end:* the upper limit of the possible hyperparameter values
    * *delta:* the step size to take through the possible parameter value range
    * *nfolds:* number of folds to partition the data set into (ie nfolds-cross validation)
  - **Functions: Constructor:**
    Takes in param_range_start, param_range_end, delta, and nfolds. Uses these to set its member variables.
    * *Destructor:* empty
    * *fitParams:* takes in a model object and uses its training data to calculate the best value for the hyperparameter in question. For KNN, it calls the function calculate_dists.fitParams then calls a setter from the model to store the best hyperparameter as a member variable for the specific model.
    * *calculate_dists:* takes in a matrix of training data and returns a matrix of distances from each example to every other example. This is specifically used for KNN.
- **GradientDescent.h/.cpp:**
  Conducts gradient descent to fit parameters for a GradientModel
  - **Variables:**

    * *iterations:* the number of iterations for gradient descent to run
    * *alpha:* the learning rate, multiplies the gradient in determining stepsize
    * *tol:* when the update norm is less than tol, the algorithm has converged
    * *batchSize:* number of examples to use in each update. 0 will use the entire sample, 1 is stochastic gradient descent. if a number greater than the sample is passed, will perform batch gradient descent
    * *costs* a vector containing the costs process of the last fit
  - **Functions: Constructor:**
    Takes in iterations, alpha, tol, and batchSize and uses them to set its member variables.
    * *Destructor:* empty
    * *fitParams:* takes in a GradientModel object and uses its training data and gradient to fit parameters in adherence to the chosen paremters. Throws and error and exits if not passed a GradientModel
    * *batchGradientDescent:* runs batch gradient descent
    * *mixedBatchGradientDescent:* runs mixed or stochastic gradient descent
    * *setBatchSize:* setter for batchSize
    * *getBatchSize:* getter for batchSize
    * *getLastCost:* gets the last cost process the model has fit