

524 Final Report: The Automatic Statistical Learner

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

January 16th, 2017

1 Introduction and scientific problem

Computer vision problems are exceptionally challenging as data processing and modelling choices are highly problem-specific. Thus approaching a new problem requires either a preliminary restriction in the direction of research, which may close off potentially valuable approaches, or an exhaustive search through a myriad of possibilities, which is extensively time consuming if done manually. The purpose of our project is to create an automated statistical learner to run such an exhaustive approach with minimal user time and manual effort required. The user will input labeled sets of training and testing data in the form of images. Our software will then test various combinations of data processing techniques and machine learning models. The system will then offer a variety of model diagnostics including a specific recommendation for a data processing technique and model combination, allowing the user to select those deemed relevant and proceed in a more-defined direction. Given the universality we intend our product to have, we have implemented the most standard computer vision data processing methodologies and the most standard statistical and machine learning models. In our directions for further research we will discuss incremental additions which could be made for more advance uses.

2 Program structure and design choices

The program consists of two key steps – data processing and model fitting. In the first step, the raw data is processed in several different ways. Then, each available model is fit to all the variants of the original data. The program outputs various diagnostics for each combination of processed data and model grouping, along with the fitted model objects for the user to use. The user, however, does not have access to these modules. Instead all user-controlled parameters are passed through the input module and reflected in the output module. In the system flowchart the user only has access to the gray boxes.

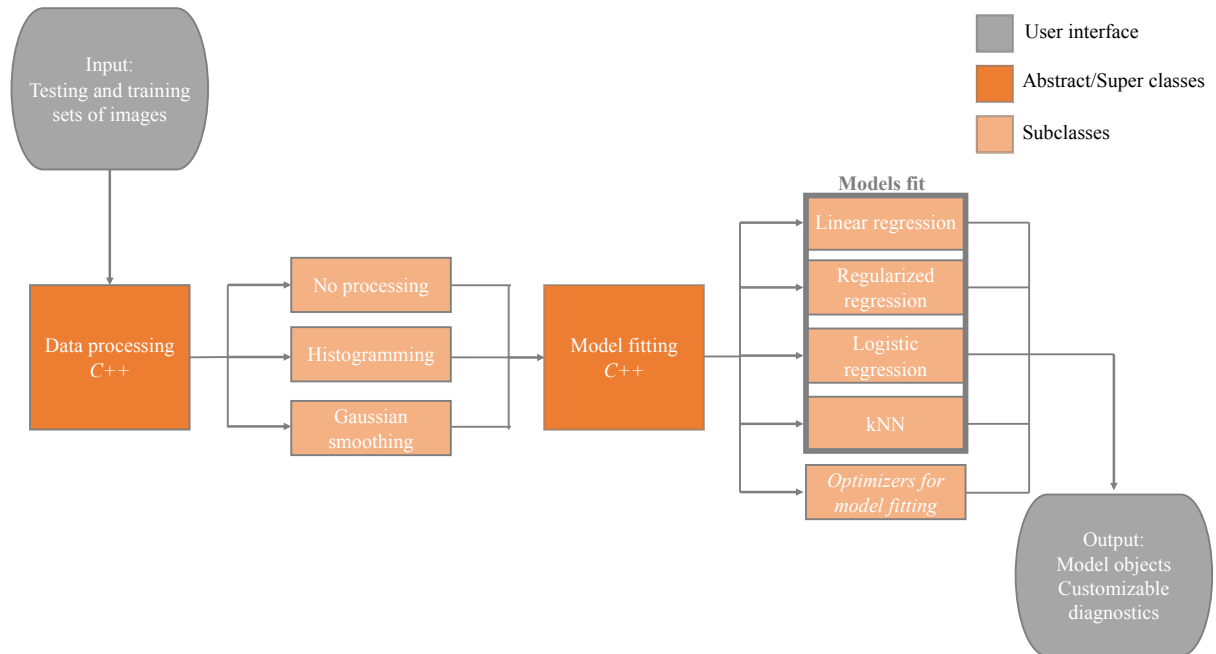


Figure 1: System Flowchart

2.1 User interface

The goal of the interface is to give the user maximum flexibility without needing access to the under-the-hood mechanics of the system. The user will interact with the software at the input and output stages exclusively, with all the option selection taking place at the input stage. Thus, while the user can control which combinations of pre-processing techniques and models are run, they do not have direct access to the implementation of these processes. They are, however, expected to have a basic knowledge of programming if they want to utilize the model objects that are given as part of the object. How they use the fit model objects is entirely up to them.

- **Input:** The input from the user should in the case of dx3-ubyte/idx1-ubyte filetypes simply consist of filenames and filepaths, where .ppm requires directory path and label filenames/paths. Each directory should represent a class label - for example “faces” and “non-faces”. Within each directory there should be defined training and testing subsets. Independent of which filetype is loaded, a datafile containing labels for the images in the directories is always needed. The initial testing was completed with the MNIST dataset¹ in which all images and corresponding labels are contained in only two files. For other datatypes than

¹<http://yann.lecun.com/exdb/mnist/>

dx3-ubyte/idx1-ubyte as in the MNIST set the program expects a directory containing multiple image files. The software also includes optional input that allows the user to select which combinations of DPMs to run. If the user is already aware that certain avenues of investigation will not be fruitful, they can run the process more efficiently by excluding these avenues.

- **Output:** The program outputs standard fitting diagnostic statistics for each of the model processing and fitting combinations. The output of model objects is, however, flexible depending on the user's initial selection for preprocessing and model fitting. The model objects enables the user to make predictions for any new data without having to re-fit the model. They will, however, still need to process their new data before feeding it into the model, for which we will provide instructions as part of the output.

Among the diagnostic statistics, total accuracy and accuracy for individual classes, error type I and error type II were computed. Where the type I error is the incorrect rejection of a true hypothesis (a "false positive"), and the type II error is incorrectly retaining a false hypothesis (a "false negative"). In this sense, a type I error is detecting an effect that is not present, while a type II error is failing to detect an effect that is present. The diagnostics of the specific predicted labels for the testing data as well as a graphical comparison of the gradient descent cost is also provided. At the end, based on the diagnostics, the optimal pre-processing and model fitting combination is suggested for the user, as determined by total testing accuracy.

2.2 Data processing

The quality of a computer vision model is directly related to the techniques used for image processing. Our most basic technique is to simply read the images as is – ie no processing. Additionally we utilize two standard computer vision techniques - Gaussian smoothing and histogramming. Gaussian smoothing simplifies the image by "smoothing" out the details. This should help avoid overfitting in the modelling step by removing extraneous details retaining only the fundamental structure of the image. Histogramming creates a histogram of the pixel values in each image. Again this removes extraneous details to assist with model fitting down the road.

2.2.1 Class structure and design choices

The data processing files all stem from the virtual class `data_process_base`. Each data processing technique has its own class. Each instance of the class contains, initially, the unprocessed data which is overwritten with the processed data after the processing function is run. This allows for a more memory efficient system. Since no processing, even after its process function is run, will always contain an unprocessed copy of the data, the other classes do not need redundant copies. The no processing class allows the user to always retain a copy of the original data just in case. The three classes have the same structure - the only difference is the implementation of the process function which warrants a more in-depth discussion.

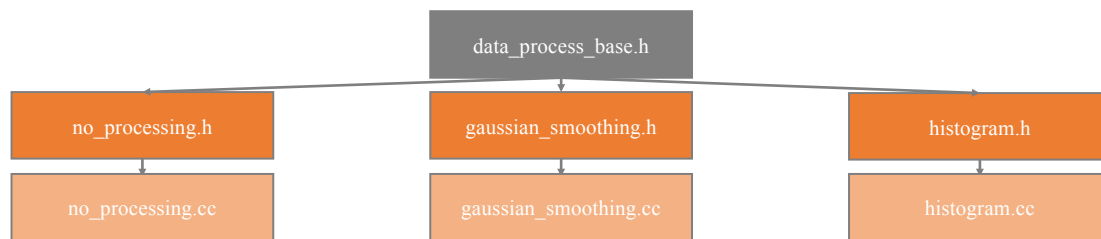


Figure 2: Data Processing Class Structure

The process function does not take inputs to ensure the user does not input incorrect references and, thus, cause the wrong data to be processed. Instead the class contains internal flags for both training and test data. When the processing function is first run, training is the processed and the flag is enabled. The second run of the processing function, repeats this process for test. Any further runs of the processing function will not throw an error, as this is not needed, but will also not do anything as accidentally processing the data multiple times will make modelling futile. The no processing and histogramming process functions are straightforward - the Gaussian smoothing is more detailed.

For Gaussian smoothing we use a standard 5x5 filter. We use a sliding window approach. Since the Gaussian filter is separable - we save memory by calculating horizontal and vertical convolutions separately using 5X1 and 1X5 calculations. This saves us from needing a 5X5 array. One way to program this is to save all horizontal convolutions in a separate array and then taking horizontal convolutions and writing them back into the original. However, to avoid requiring an extra array we calculate horizontal and vertical convolutions simultaneously - we store 5 rows of horizontal convolutions across the image. For border conditions we use mirroring both along the sides and diagonally at the corners.

Each class contains accessor and modifier functions to allow the imported data to be fed into each class. The accessor functions all return vectors of armadillo matrices for the modelling class. No processing and Gaussian smoothing both return vectors of the same dimension as the original since they return the original images - of course, in the Gaussian case these are smoothed versions of the original images. The histogramming function returns an extra image along with the vector - this extra image is the histogram. Within the driver file the histogram is then converted to a vector of single row armadillo matrices.

We made the algorithms in house. We used armadillo to ensure compatibility with the remainder of the system and the standard library to avoid unnecessary redundancies.

2.3 Model fitting

After the input data is processed and standardized, our software will use it to fit and evaluate several possible models. We choose to implement linear and logistic regression, regularized regression (encompassing lasso, ridge, and elastic net regression), and k-nearest neighbors, for their

general simplicity and overall popularity. We will use a generalized gradient descent algorithm (which allowed for batch gradient descent, stochastic gradient descent, and mixed batch gradient descent) and cross validation to optimize parameter selection to fit the best version of each model to the training data. Because we cannot assume that the user will provide an unbiased selection of training and testing data, we employed randomized selection to create new partitions, and reported generalized diagnostics in addition to diagnostics on the intended train-test partition.

2.3.1 Class structure and design choices

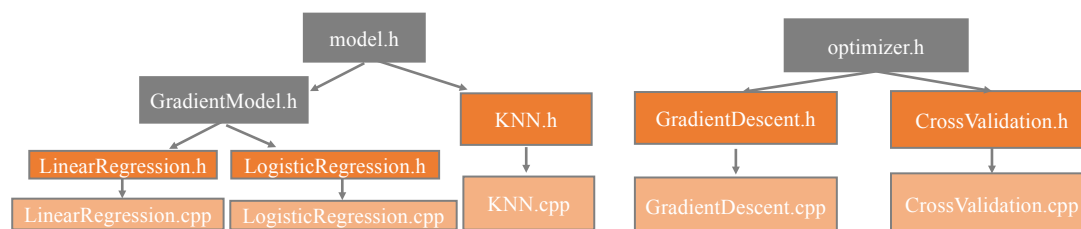


Figure 3: Model Class Structure

Because each model is fit using an optimization algorithm, we decided to have two broad abstract classes—model and optimizer. A model object takes in training data, training labels, and an optimizer object, which it uses to fit the model. The optimizer is passed the model object, so that it can access public methods and variables specific to that model.

The model class includes functions general to models: fitting the model, getting the training data and labels, a constructor, a destructor, and a method that predicts labels for testing data. These getters and setters are necessary for the general optimizer class to fit these models. The three models that we implemented are k-nearest neighbors (KNN), linear regression, and logistic regression. We extended the Model class to another virtual class, GradientModels, to allow access to a gradient method for use in gradient descent. While KNN only requires optimization of a hyperparameter, regression requires a gradient in order to fit it via gradient descent. Therefore, we included another abstract class under model called GradientModel. Linear regression and logistic regression are separately implemented as their own classes.

Linear regression has optional arguments for ℓ_1 and ℓ_2 penalty terms, facilitating regularization. If ℓ_1 and ℓ_2 are uninitialized, they are set to 0.0, ensuring no regularization. Positive ℓ_1 and $\ell_2 = 0.0$ corresponds to the machine learning model known as the lasso, $\ell_1 = 0.0$ and positive ℓ_2 to ridge regression, and both positive is the elastic net. The gradient descent class, in conjunction with the flexible `LinearRegression::Gradient` method, is capable of fitting all of these, and the `get_exact_params` method can be used to check that the model is fitting correctly, as both the ridge and standard linear regression cases are solved in closed form.

The optimizer class includes a common constructor, destructor, and function that takes in a model and fits its parameters. We implemented gradient descent, used to fit linear and logistic regression, as well as cross validation, used to select the hyperparameter k for KNN. The Cross-Validation class performs 10-fold cross validation by default and allows the user to select an appropriate number of folds. One of the primary issues in fitting gradient descent is determination of the learning rate, α . For batch gradient descent (ie using the entire training data), a well known method of doing so is implementing an evolving learning rate through a method called bold driver, in which the learning rate increases when the optimizer changes the parameters in a way that is beneficial and decreases the learning rate when the optimizer takes a step in the wrong direction. For stochastic and mixed batch gradient descent, we left it to the user to determine learning rate, as methods for fine tuning the learning rate were beyond the scope of this implementation. For a naive user, would we recommend utilizing batch gradient descent.

To enable tractability for the addition of further models and optimizers in the future, it was important to keep the model and optimizer virtual classes as general as possible. However, we faced the constraint of GradientDescent being solely applicable to GradientModels. To counteract this, we utilized polymorphism to make the virtual method `Optimizer::fitParams` adapt to the specific models in question. Certainly, this approach would be greatly utilized should we extend the scope of our project.

We have implemented this module in C++ because of the anticipated speed bottleneck in fitting models to large data sets. We took advantage of the linear algebra C++ library Armadillo to work with the matrices of data. We chose to implement the optimization methods and models ourselves rather than use existing libraries. This allowed us to use our own customized data classes, enabling us to only include necessary methods and variables for computer vision classification.

3 Development process

3.1 Planning

Due to the short time frame of the project it was a challenge to keep the already tight schedule seen in our Design Document. The short time frame did however prove the strength of Github as a platform for coding projects among collaborators who are physically apart. As the project progressed with time our initial schedule was revised as details/modules were revised.

3.2 Unit testing procedure

The unit testing of the Automated Statistical Learner strives to be intuitive to implement, easily expandable, and modular. To achieve the first goal the choice of Python was made over tools such as Google Test, Boost, or CppUnit. The python script runs the compiled C++ executable

and thus is a completely separate entity from the main C++ code. This simplifies the testing procedure, especially from a team development perspective, as there are no make files to update or dependencies to worry about. As a tester, all that is needed is for the latest *working* executable to be available; the development team is free to add a new feature while the testing team works on verifying the latest working version. This streamlines the development cycle and is a helpful separation of duties: the unit testers can often have fresh ideas that the developers did not think of because it was “their” code.

Using Python’s unit testing module, the testing is very simple to expand. For example, if another image type is to be imported (e.g., PNG or TIFF), then a simple suite of tests can be added through Python’s unit testing syntax. The work flow that is unit tested is divided into two main sections: input and modeling, for which there are different types of unit testing needed for each. For input, the majority of the unit testing is to handle whether the data was successfully imported, whether all the pixel values are valid, and if it was processed correctly. With Python it is simple to add more tests as needed, for example, if the team were to wish to unit test the outputting and plotting routines. However for this implementation it was decided that the plots did not need to be unit tested.

Given the choice of unit testing framework, the most challenging aspect of the unit testing was ensuring modularity. In other words, making sure we only run the pieces of the executable that we want to test. Since there is no use of SWIG/Cython, Python’s communication from C++ is via standard out, which it pipes in as a string and then parses. By carefully limiting the standard out and only outputting during a unit test, this is not a performance loss. In order to determine which portions of the code to run, Python passes a series of flags to the executable. Since the program has three stages: input, modeling, output, this structure is easily implemented and maintained. In addition, these flags can be interpreted by C++ to purposely create error conditions, so that Python can check if an error has occurred. For example, in the during the check for proper pixel values, a unit testing flag is passed which tells C++ to artificially flip one pixel to a value of -1. Then Python ensures that there is indeed an assertion that is caught, otherwise it fails the unit test. This type of artificial testing allows unit testing without necessarily needing to do “heavy lifting” in data import or processing.

Note: The unit testing was designed to be run with `alpha_driver` which is the “developer oriented” version of the main driver. The main driver is more user friendly - oriented to those who do not need an “under the hood” look.

3.3 Version control

As we build the interface and modules, we continuously tested and debugged to ensure that each incremental change was error-free. Our preliminary plan of constructing one working, thoroughly debugged, pass through the system, utilizing the simplest possible options proved to be successful. No data processing for the first step and simple linear regression in the second. We successfully established a paradigm of horizontal communication and used this as a baseline for testing as new classes were added to pre-processing and model fitting. New pre-processing techniques

were tested with the previously finalized regression model fit. For each of the two tests separate unit tests were implemented for problems specific to those processes.

Version control took on two forms. First, GitHub was used to assure all versions were saved and incremental changes were properly catalogued. Second, the project in its nature is modular and was implemented as such: with parallel branches containing the development of the different modules.

The modularity and clearly defined group roles meant that every user was responsible for a separate piece of the system. Each user held writing privileges to his/her respective branch with the understanding that the other branches were not to be edited without consulting the group members responsible. Once the users' code needed feedback, a pull-request was to be created which will allowed the other users to examine the written code. This process was repeated until the branch was ready for deployment and merged with the master branch. Before acceptance, the deployment allowed for verification of the code by all parties and roll-back if necessary. Whenever an additional model or data processing method was added, we tested for compatibility with all previous finalized classes. This modular building minimized the number of versions of each piece needed as the project developed.

3.4 Alpha vs. beta

The objective for the alpha version was to create a working pass through the system with one form of processing and one form of modeling. The beta built on on this existing skeleton, by adding classes to both modules as well as expanding input and output capabilities.

3.4.1 Importer

Since the goal for the Alpha version was to complete a single pipeline through the system only one type of data importer was needed. This first importer could only take in the data-formats dx3-ubyte/idx1-ubyte which among others is used by the MNIST data-set.

For the Beta version a .ppm image importer was coded since this is a wide-spread data-type for machine learning data-sets. In order to keep the importer methods compatible with the data processing and model fitting we agreed on a standard format for the loaded images. The images were loaded pixel by pixel into Armadillo matrices which were arranged in vectors. The labels was likewise consistently loaded into vectors matching each Armadillo matrix in the vector holding these. The agreement to make this a standard made it more manageable to code a new importer since the format was already known and agreed upon.

3.5 Data processing

The main focus for the alpha was to build a class structure which would be compatible with the model fitting team and would be easily reproducible for incremental processing types. We

wrote `data_process_base` and the `no_processing` class to establish this. For the beta version we added Gaussian smoothing, histogramming, and all of the unit tests. As we had the class structure established from the alpha, our largest challenge in implementing the beta was compatibility with that class structure. Specifically, histogramming actually returns an object of an entirely different dimension than the other two methods. Thus, we had to have a somewhat customized approach where, in the driver, we further reformat the data.

3.5.1 Model Fitting

For the alpha version, we implemented the abstract model and optimizer classes, as well as the header and .cpp files for linear regression and gradient descent. This way, we had a model and a mechanism to fit it, so we were able to test a path through the program. For the beta version, we added logistic regression, KNN, cross validation for KNN, and updated gradient descent. In between the beta and final versions, we added the abstract `GradientModel` class as well as generalized cross validation to select hyperparameters for `GradientModels`.

4 Results/Sample output and concluding statements

4.1 Data processing



Figure 4: Original image/No processing



Figure 5: Gaussian smoothing

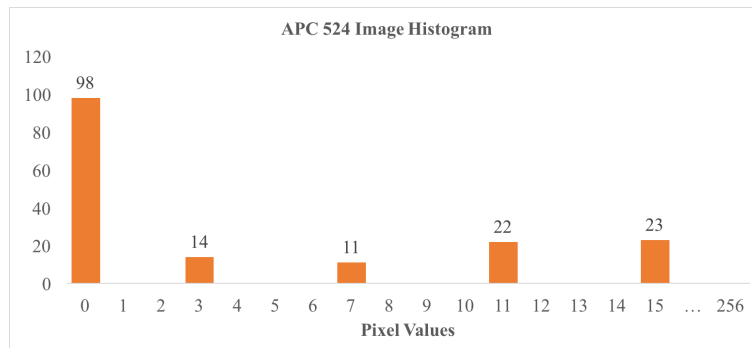


Figure 6: Histogram

4.2 Model fitting

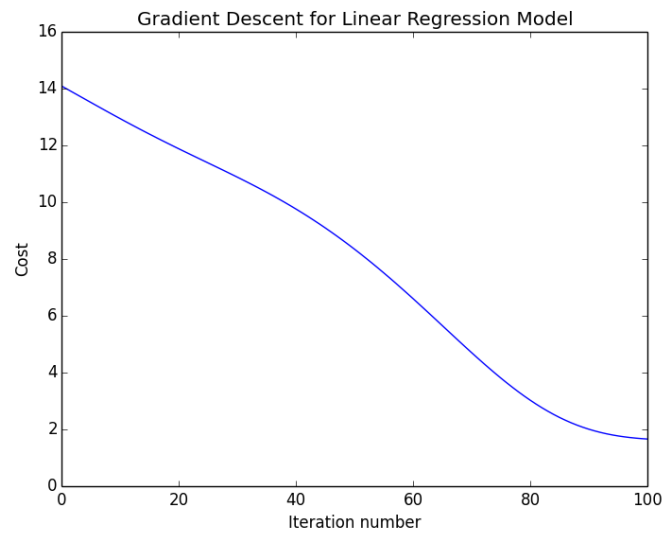


Figure 7: Gradient Descent for Linear Regression Model

```

Results for k-nearest neighbors on gaussian smoothing data:

For label 0, the class testing accuracy is 0.986486
For label 0, the test frequency of type 1 error is 0.0104654
For label 0, the test frequency of type 2 error is 0.0135135

For label 1, the class testing accuracy is 0.984444
For label 1, the test frequency of type 1 error is 0.0123909
For label 1, the test frequency of type 2 error is 0.0155556

For label 2, the class testing accuracy is 0.858052
For label 2, the test frequency of type 1 error is 0.00753558
For label 2, the test frequency of type 2 error is 0.141148

For label 3, the class testing accuracy is 0.894608
For label 3, the test frequency of type 1 error is 0.0139159
For label 3, the test frequency of type 2 error is 0.105392

For label 4, the class testing accuracy is 0.856459
For label 4, the test frequency of type 1 error is 0.00948925
For label 4, the test frequency of type 2 error is 0.143541

For label 5, the class testing accuracy is 0.825269
For label 5, the test frequency of type 1 error is 0.00826674
For label 5, the test frequency of type 2 error is 0.174731

For label 6, the class testing accuracy is 0.92328
For label 6, the test frequency of type 1 error is 0.00717637
For label 6, the test frequency of type 2 error is 0.0767196

For label 7, the class testing accuracy is 0.824818
For label 7, the test frequency of type 1 error is 0.00807799
For label 7, the test frequency of type 2 error is 0.175182

For label 8, the class testing accuracy is 0.869792
For label 8, the test frequency of type 1 error is 0.0157589
For label 8, the test frequency of type 2 error is 0.130208

For label 9, the class testing accuracy is 0.90051
For label 9, the test frequency of type 1 error is 0.026046
For label 9, the test frequency of type 2 error is 0.0994898

The overall testing accuracy for k-nearest neighbors on gaussian smoothing data is 0.892777

```

Figure 8: Testing Accuracies and Error Types by Class

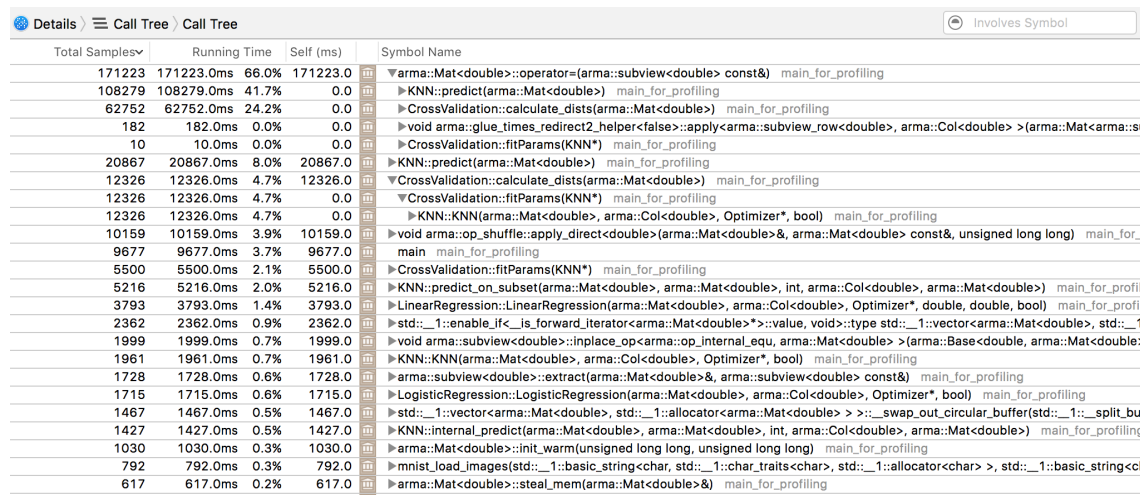
4.3 Profiling

Our main goal in profiling was to identify the bottlenecks in our code and attempt to make those sections more efficient. We used Instruments, part of Apple Developer, to conduct profiling. We mainly focused on the counters profiling instrument, which collects stack trace information from performance monitor counters at regularly spaced intervals—the main idea being that if a function is sampled an excessive number of times, there is likely a mechanism surrounding that portion of code that if made more efficient can greatly impact our software’s performance.

It was simple to identify ways to make several of the functions with the largest numbers of samples more efficient. For example, the KNN algorithm requires the distance from one sample to all others. Utilizing memoization, we were already only computing each distance once, storing it, and then accessing it. However, in the double for loop to calculate and fill out a distance matrix, we were accessing and storing the example corresponding to the outer for loop in the inner loop. Clearly, this is wasteful when we can merely do this once before the inner loop, since the variable stays the same. Although we should have realized these types of inefficiencies initially when coding the algorithm, profiling helped us open several of these inadvertent bottlenecks. As you can see in figure 7, even with this speed-up, the calculate dists function in Cross Validation is still one of the most commonly sampled functions.

The figure below contains an example Instruments output from running our main program.

Total samples are simply the count of the number of times the function was sampled; self is the time in miliseconds spent in that function, excluding other functions it calls, and running time is the amount of time the symbol ran along with the percentage of total time that it occupied. The most-sampled function is an internal Armadillo function, called from KNN's predict function and Cross Validation's calculate distance function through its fitParams function, that appears to find a submatrix view (ie, a slice of a matrix). 41.7% of time in the program is spent in this submatrix Armadillo function within KNN's predict function. Because finding the k nearest neighbors to each test example does not depend on other example's nearest neighbors, this portion of the code is a fantastic candidate for massive performance improvements in terms of parallelization. We discuss our limitations with parallelization in the future work section. Through profiling our code, we were able to identify and troubleshoot several performance bottlenecks.



Total Samples	Running Time	Self (ms)	Symbol Name
171223	171223.0ms 66.0%	171223.0	▼arma::Mat<double>::operator=(arma::subview<double> const&) main_for_profiling
108279	108279.0ms 41.7%	0.0	►KNN::predict(arma::Mat<double>) main_for_profiling
62752	62752.0ms 24.2%	0.0	►CrossValidation::calculate_dists(arma::Mat<double>) main_for_profiling
182	182.0ms 0.0%	0.0	►void arma::glue_times_redirect2_helper<false>::apply<arma::subview_row<double>, arma::Col<double>> (arma::Mat<arma::s
10	10.0ms 0.0%	0.0	►CrossValidation::fitParams(KNN*) main_for_profiling
20867	20867.0ms 8.0%	20867.0	►KNN::predict(arma::Mat<double>) main_for_profiling
12326	12326.0ms 4.7%	12326.0	▼CrossValidation::calculate_dists(arma::Mat<double>) main_for_profiling
12326	12326.0ms 4.7%	0.0	▼CrossValidation::fitParams(KNN*) main_for_profiling
12326	12326.0ms 4.7%	0.0	►KNN::KNN(arma::Mat<double>, arma::Col<double>, Optimizer*, bool) main_for_profiling
10159	10159.0ms 3.9%	10159.0	►void arma::op_shuffle::apply_direct<double>(arma::Mat<double>&, arma::Mat<double> const&, unsigned long long) main_for_
9677	9677.0ms 3.7%	9677.0	main main_for_profiling
5500	5500.0ms 2.1%	5500.0	►CrossValidation::fitParams(KNN*) main_for_profiling
5216	5216.0ms 2.0%	5216.0	►KNN::predict_on_subset(arma::Mat<double>, arma::Mat<double>, int, arma::Col<double>, arma::Mat<double>) main_for_profiling
3793	3793.0ms 1.4%	3793.0	►LinearRegression::LinearRegression(arma::Mat<double>, arma::Col<double>, Optimizer*, double, double, bool) main_for_profiling
2362	2362.0ms 0.9%	2362.0	►std::__1::enable_if<__is_forward_iterator<arma::Mat<double>*>::value, void>::type std::__1::vector<arma::Mat<double>, std::__1
1999	1999.0ms 0.7%	1999.0	►void arma::subview<double>::inplace_op<arma::op_internal_eu, arma::Mat<double>> (arma::Base<double, arma::Mat<double>
1961	1961.0ms 0.7%	1961.0	►KNN::KNN(arma::Mat<double>, arma::Col<double>, Optimizer*, bool) main_for_profiling
1728	1728.0ms 0.6%	1728.0	►arma::subview<double>::extract(arma::Mat<double>&, arma::subview<double> const&) main_for_profiling
1715	1715.0ms 0.6%	1715.0	►LogisticRegression::LogisticRegression(arma::Mat<double>, arma::Col<double>, Optimizer*, bool) main_for_profiling
1467	1467.0ms 0.5%	1467.0	►std::__1::vector<arma::Mat<double>, std::__1::allocator<arma::Mat<double>> > >::swap_out_circular_buffer(std::__1::__split_bu
1427	1427.0ms 0.5%	1427.0	►KNN::internal_predict(arma::Mat<double>, arma::Mat<double>, int, arma::Col<double>, arma::Mat<double>) main_for_profiling
1030	1030.0ms 0.3%	1030.0	►arma::Mat<double>::init_warm(unsigned long long, unsigned long long) main_for_profiling
792	792.0ms 0.3%	792.0	►mnist_load_images(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>>, std::__1::basic_string<cl
617	617.0ms 0.2%	617.0	►arma::Mat<double>::steal_mem(arma::Mat<double>&) main_for_profiling

Figure 9: All functions (excluding system libraries) with at least 500 samples collected by Instruments. The call tree is inverted.

4.4 Division of work

Due to the modularity of our project it was very feasible to divide it into the parts described above. Every group member was responsible for a specific part of the project while we made ourselves available to assist each other whenever there was excess time. Agreements on standards for input and output resulted in better compatibility between the different modules and made work division easier.

4.4.1 Data importer

Bill designed the initial data importer for MNIST, from which the structure (input/output scheme) for the other importers was based. Andreas worked on the image and label importers for other

filetypes: .ppm, and .jpg.

4.4.2 Data processing

Nina and Bill worked on data processing with Nina writing the algorithms/class structure and Bill writing unit testing. The vast majority of the work - debugging, harassing Jeffry, and Git committing - was done by Bill and Nina together.

4.4.3 Modeling

Chase and Ari wrote the optimizers and model classes. They coded the abstract classes and linear regression together. Chase individually coded gradient descent, logistic regression, and regularized regression, and Ari coded KNN and cross validation. Chase did unit testing for this portion.

4.4.4 Main driver

Chase wrote the main driver.

4.4.5 Output

Noemi and Andreas worked on some python interface and reporting that were no longer used given the need of a full c++ implementation. Worked on the performance module, customizing output, plotting graphs.

4.4.6 Profiling

Ari conducted all profiling.

4.4.7 Project manager

Nina was project manager and organizer for written submissions and presentations.

4.4.8 Github master

Ari maintained admin privileges on the Github shared directory and managed submissions, pulls, etc.

4.4.9 User manual

Noemi created/formatted our user manual and wrote the input portion. Nina and Chase filled in the appropriate sections for data processing and modeling respectively.

4.5 Roadblocks and lessons learned

The key challenge in this process was both internal and external compatibility. Internally, the challenge was in the combination of the various pieces of the system. Due to time limitations the separate pieces of the software were written simultaneously. Naturally, as code is written it diverges away from the original blueprint to allow for previously unforeseen errors and challenges. Such evolutions had to be communicated throughout the system and compatibility had to be maintained at each step to ensure that the final products from one step would smoothly feed in as the inputs to the next. Since we used inheritance a decent amount throughout our code we had to contend with the design challenges of adding incremental modules in a consistent manner. Histogramming, for example, returns a separate object than the original data. The processing classes, however, were designed to overwrite the original data with the processed data to be memory efficient. Thus, the histogramming process function actually adds an additional matrix to the original data - this matrix being the histogram. Then in the driver programs this matrix is reshaped into a vector of single row arma matrices. Given more time, we would have perhaps reconsidered a way to make a more efficient class structure to work around such conflicts. At the end of the day, however, no class structure will be flexible enough to account for all possibilities.

Externally we experienced difficulties working with Windows vs. Macs for personal computers as well as certain libraries being outdated on the clusters vs. our personal machines. For example, we wanted to use Allinea MAP for profiling, but we (and CSES) were unable to get our code to run correctly on Nobel, even though it compiled. Similarly, we were unable to parallelize the model fitting portion of the program because we couldn't get OpenMP to work on our macbooks and the program wouldn't run on Nobel.

Originally, we had planned to utilize Python for the first half of the system - input and data processing. Python is generally more user friendly and several of our team members had more experience with it. Then we had planned to utilize Swig or Cython to combine the two parts of the system. We explored the use of Cython, but discovered Cython is generally more conducive to improving runtime of individual functions within Python code by using C++. Given that a significant portion of our system required C++ and would be outputting results in C++ - since the model objects to be used for prediction are C++ objects - we concluded Cython would not serve our needs. Given this and the fact that certain of the data processing functions could also become quite memory inefficient, led us to decide to code our entire program in C++.

4.6 Conclusions and directions for continued research

In conclusion, we achieved our planned automatic statistical learner. Along the way we had to make a significant structural change to our design - writing the entire system in C++ rather than Python and C++. Albeit this created a rather steeper learning curve for those less familiar with C++, in the end, this improved compatibility, speed, and the substructure of the data processing

modules. Our system achieves the stated goal of providing the user with summary outputs on a variety of data processing and model combinations.

Directions for continued research:

While our system offers a significant improvement in efficiency over manually coding and running various combinations, it is of course not entirely universal in scope. Continued development would allow for our system to run further data processing techniques and models. Currently, our data processing techniques do not distinguish between different structure within the image - for example, edges vs. interior. Next steps would be to develop such techniques.

Models we considered but lacked the time to implement include SVM, various kernel models, naive Bayes, and neural networks. While the latter of these is highly effective in computer vision applications, the sheer scale of the task and the flexibility required to develop nets of various shapes and sizes was beyond the scope of this project. Additional model fitting algorithms could also improve parameter fitting for especially unpleasant and nonconvex problems. In addition to including more efficient algorithms in this portion of the code, we would ideally parallelize our code using OpenMP due to its simplicity.

The front end of the system is currently adapted to take in some of the standard types of files for these problems (ppg, jpg, and the MNIST data set); however, this step could be expanded and diversified to include filetypes such as png as well. A further step would be to permit the user to directly interact with the various diagnostics and to even add to the system on his or her own.