CSI5149 Project 2 Report - Classification

Shaughn Finnerty
6300433
March 27, 2016

Running Instructions

Install Dependencies

To run this project, you will need the following dependencies install

- **Python 2.7**
- **numpy** for matrix operations (i.e. computing the transpose of feature matrices and parameter matrices)
  - `pip install numpy`
  - http://docs.scipy.org/doc/numpy-1.10.1/user/install.html
- **scipy** for optimization methods (i.e. minimizing the cost function in the logistic regression classifier)
  - `pip install scipy`
- **tabulate** for pretty printing CSV result files
  - `pip install tabulate`

As seen the pip package manager for python is the best tool to install these dependencies. This package manager can be installed by following the instructions here @ https://pip.pypa.io/en/stable/installing/ (downloading a python file and executing it)

When this is downloaded you can use the `requirements.txt` file to handle all the dependences that need to be installed by running the following command from within the project directory:

`pip install -r requirements.txt`

If you have already ran the pip install commands for each dependency separately, there is no need to perform this command.

Note: We have used numpy and scipy only for the functions that would otherwise be available natively in MatLab (transposition, minimization) and requested permission from the professor before attempt. These are only used in the logistic regression classifier. We have not utilized pre-existing classification algorithms from libraries like Scikit-Learn.

Running Instructions

To run the project and generate results for the K-Nearest-Neighbour classifier, run the following command:

`python run.py --knn`

This will train and test the K Nearest Neighbour classifier on all datasets. Furthermore, you can evaluate the other classifiers by running:

`python run.py --gnb`

`python run.py --kde`

`python run.py --lr`

To run the Gaussian Naive Bayes classifier, the Kernel Density Estimate Naive Bayes classifier, and the Logistic regression classifier, respectively. To run ALL of the classifiers one after the other, use the command:

`python run.py --all`

The project has come with the CSV results in the `./results` directory in case you decide not to run them all (as some classifiers will take time to train and test, especially on the larger datasets).

Note: The Gaussian Naive Bayes algorithm will execute in the fastest time. You may will need to wait a fairly extended time for the other algorithms to train and test their models.

Description & Discussion of Model & Algorithms

K-Nearest-Neighbour

With K-Nearest-Neighbour, the initial model is created from the training data (e.g. 80% of the observations in the provided dataset) by simply transforming the matrix of n features * m observations/items to an easily accessible list/array of vectors, each of which containing the data points for each m features, and the class to which the observation belongs. While there is no parameters to be learned in this model (since it is purely memory-based), the transformation of data allows the list of vectors to be iterated for every classification required in the test phase.

To classify a new observation with "unknown" class, we simply take the feature values for that item in a vector and compute the euclidian distance (note: other distance measures can be used) to the vectors initially created for each observed/training item available. We then sort the list of vectors based on increasing distance. The intuition behind this is that vectors in the training observations that have feature values closer (smaller distance) to those in the unknown item to be classified are likely to have a similar class to the unknown. We then take the first k closest training items (e.g. in our simulations we chose 51), and count the items that belong to class_1 and those that belong to class_2. The class having the most items in the k nearest neighbours is then assigned to be the class of the unknown item (majority vote). Note: Since we are using binary classification, it is useful to choose an odd K to avoid a case where there are an equal amount of items belong to both classes in the first k neighbours.

As you will see, this model and algorithm led to the best classification results, achieving > 90% accuracy on all datasets. As such I chose to highlight this algorithm as the **best algorithm** for the project as it provided the best accuracy.

The one shortfall was that on large datasets (with large amounts of neighbours to compare to and distances to compute), the computation can take a long time. In future work it would be interesting to experiment with feature selection. That is, as part of the training process, we could identify features that have the most "influence" when determining the class, and use only these features when computing the distance. Although this would not reduce the amount of neighbours that need to be compared, it would reduce the amount of computations that need to occur when calculating the euclidian distance between the vectors as we would only need to account for the best selected features.

Before implementing the KNN classifier, other models/algorithms were implemented in hopes to achieve desired accuracy, but did not prove to perform as well as KNN. However, it is a useful exercise to report on the models/algorithms implemented.

Gaussian Naive Bayes

In this generative model/algorithm, we use a Bayesian approach to produce a function: $P(X|Y)$ where X is the observed item, and Y is the class. This is achieved in the training phase by assuming that each feature in each class in the training items are identically and independently distributed according to a Gaussian Distribution. To create this model, we first calculate the mean and standard deviation for each feature in each class (using values observed from each training item in their respective class). With these parameters, we can then classify/test new unknown items by computing the product of the values of the Gaussian Probability Density Function for each feature in each class. For example to compute the probability that the unknown item X belonged to class 1, we calculated:

$$P(X = x|Y = class1) = \prod_{i=1}^{N} N(x_i, \mu^i_{class1}, \sigma^i_{class1})$$

Where N is the amount of features which were all assumed to be iid according to a Gaussian distribution for each class.

A similar probability was calculated for class 2. Of the two, the class with the highest probability calculated was the class to which the unknown item was assigned.

Kernel Density Estimate w/ Naive Bayes

It was thought that the features might not be distributed normally, so we hoped that we could compute a Kernel Density Estimate of the continuous distribution for each feature in each class might improve the performance. This was performed by computing the average of Gaussians with mean set to every possible continuous value for a given feature observed in the training data (so multiple Gaussian density functions had to be evaluated for each feature, rather than a single one using the computed mean and standard deviation in training). That is, given a value x_i, its density estimate for class 1 became:

$$\frac{1}{M} * \sum_{j=1}^{M} N(x_i, u_i = x_j, \sigma_{class1})$$

Where M is the amount of observations for the feature whose continuous distribution is being estimated with a Gaussian kernel.

This density estimate was then used in the product computed similar to above in the Gaussian Naive bayes classifier for each feature to compute a probability for each class. The selected class was determined by a similar distinction (i.e. the product of estimated distribution functions that evaluated to provide the highest probability). This technique to use a kernel density estimate of the continuous distribution for each feature was inspired by the Flexible Naive Bayes model presented by John and Langley (1995).

Unfortunately, we did not see any noticeable increase in performance using this kernel density estimate for the distribution of each feature in Naive Bayes. One possible hypothesis is that the data for each feature was in fact normally distributed, and thus, the estimation yielded a distribution that was very similar. This would explain why the accuracy results are very similar between the tests run for each algorithm.

The results from this algorithm did not show any significant benefit to using the algorithm. In fact, since we had to compute Gaussians for every feature value, and store these feature values as parameters (John and Langley, 1995), the increased storage and processing required made it less favorable than the Gaussian Naive Bayes for this particular problem (where we just had to store and process the initially computed expected values and standard deviations for each feature).

Logistic Regression

The logistic regression classifier allowed us to experiment with creating a discriminative classifier. Compared to a generative classifier with the Naive Bayes, this

models the conditional probability $P(Y|X)$ of a class y given an observed item to be classified x.

In a logistic regression classifier, our hypothesis representation is defined using the sigmoid function with the product of our transposed parameter vector and feature vector as:

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

And since we are solving a binary classification, our model then becomes:

$$P(Y = class2 | X = x_i) = h_\theta(x_i)$$
$$P(Y = class1 | X = x_i) = 1 - h_\theta(x_i)$$

To fit/estimate the parameters, Θ, for our hypothesis function, we define a cost function that will be minimized using the training data. By minimizing this cost function over all parameters, we get the parameters which result in the least amount of error for our hypothesis function on our training data. Depending on the true value of the class (y) for a given training item, the cost function is defined like so:

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

As you can see, the cost function increases when our hypothesis function provides an hypothesized class value that is further away from the true (known) class of the item.

This cost function is used on all items in the set of training data to estimate all parameters $\theta_i \in \Theta$ for each feature. When training, we use the `scipy.optimize.fmin` method to minimize the following function and provide the minimized parameter vector to be used for further classification in the testing phase:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

Where m is the number of items in the training dataset.

The `scipy.optimize.fmin` function was useful for minimizing the above function and determining the best-fit parameters. This function minimizes a given function using the downhill simplex algorithm (only using function values, not derivatives or second derivatives). A useful exercise would be to implement the gradient descent method to minimize this function. However, for the sake of time, the provided `fmin` algorithm was used.

One problem that we ran into on larger datasets is that when minimizing, the parameters would converge to a point where the hypothesis function would produce 0 values within the logarithmic function used in the cost function. To account for this domain error, we added an very small epsilon value (10^{-10}) to the cost function so that it would not raise an error during execution.

Results

Each table provides a summary of testing in a 4:1 ratio of training items to testing items on the datasets provided for this project.

We have ordered the results based on which classifier performed best in the experiments.

Results from K-Nearest-Neighbour Classifier Training & Testing, K=51						
Dataset Name	Total Items in Dataset	# Of Items In Training Dataset	# Of Items In Test Dataset	Correct Predictions	Incorrect Predictions	Accuracy
classify_d3_k2_saved1.mat	2000	1600	400	381	19	0.9525
classify_d3_k2_saved2.mat	2000	1600	400	394	6	0.985
classify_d3_k2_saved3.mat	2000	1600	400	399	1	0.9975
classify_d4_k3_saved1.mat	2000	1600	400	375	25	0.9375
classify_d4_k3_saved2.mat	2000	1600	400	363	37	0.9075
classify_d5_k3_saved1.mat	2000	1600	400	389	11	0.9725
classify_d5_k3_saved2.mat	2000	1600	400	373	27	0.9325
classify_d99_k50_saved1.mat	3960	3168	792	789	3	0.99621

classify_d99_k50_saved2.mat	3960	3168	792	790	2	0.99747
classify_d99_k60_saved1.mat	3960	3168	792	790	2	0.99747
classify_d99_k60_saved2.mat	3960	3168	792	787	5	0.99369

Results from Gaussian Naive Bayes Classifier Training & Testing						
Dataset Name	Total Items in Dataset	# Of Items In Training Dataset	# Of Items In Test Dataset	Correct Predictions	Incorrect Predictions	Accuracy
classify_d3_k2_saved1.mat	2000	1600	400	333	67	0.8325
classify_d3_k2_saved2.mat	2000	1600	400	269	131	0.6725
classify_d3_k2_saved3.mat	2000	1600	400	356	44	0.89
classify_d4_k3_saved1.mat	2000	1600	400	286	114	0.715
classify_d4_k3_saved2.mat	2000	1600	400	278	122	0.695
classify_d5_k3_saved1.mat	2000	1600	400	282	118	0.705
classify_d5_k3_saved2.mat	2000	1600	400	273	127	0.6825
classify_d99_k50_saved1.mat	3960	3168	792	591	201	0.74621
classify_d99_k50_saved2.mat	3960	3168	792	578	214	0.7298
classify_d99_k60_saved1.mat	3960	3168	792	592	200	0.74747
classify_d99_k60_saved2.mat	3960	3168	792	559	233	0.70581

Results from Kernel Density Estimate Naive Bayes Classifier Training & Testing						
Dataset Name	Total Items in Dataset	# Of Items In Training Dataset	# Of Items In Test Dataset	Correct Predictions	Incorrect Predictions	Accuracy
classify_d3_k2_saved1.mat	2000	1600	400	326	74	0.815
classify_d3_k2_saved2.mat	2000	1600	400	271	129	0.6775
classify_d3_k2_saved3.mat	2000	1600	400	346	54	0.865
classify_d4_k3_saved1.mat	2000	1600	400	294	106	0.735
classify_d4_k3_saved2.mat	2000	1600	400	279	121	0.6975
classify_d5_k3_saved1.mat	2000	1600	400	289	111	0.7225

classify_d5_k3_saved2.mat	2000	1600	400	280	120	0.7
classify_d99_k50_saved1.mat	3960	3168	792	586	206	0.7399
classify_d99_k50_saved2.mat	3960	3168	792	579	213	0.73106
classify_d99_k60_saved1.mat	3960	3168	792	591	201	0.74621
classify_d99_k60_saved2.mat	3960	3168	792	561	231	0.70833

Results from Logistic Regression Classifier Training & Testing						
Dataset Name	Total Items in Dataset	# Of Items In Training Dataset	# Of Items In Test Dataset	Correct Predictions	Incorrect Predictions	Accuracy
classify_d3_k2_saved1.mat	2000	1600	400	289	111	0.7225
classify_d3_k2_saved2.mat	2000	1600	400	288	112	0.72
classify_d3_k2_saved3.mat	2000	1600	400	281	119	0.7025
classify_d4_k3_saved1.mat	2000	1600	400	295	105	0.7375
classify_d4_k3_saved2.mat	2000	1600	400	278	122	0.695
classify_d5_k3_saved1.mat	2000	1600	400	301	99	0.7525
classify_d5_k3_saved2.mat	2000	1600	400	292	108	0.73
classify_d99_k50_saved1.mat	3960	3168	792	582	210	0.73485
classify_d99_k50_saved2.mat	3960	3168	792	589	203	0.74369
classify_d99_k60_saved1.mat	3960	3168	792	601	191	0.75884
classify_d99_k60_saved2.mat	3960	3168	792	597	195	0.75379

References

Breheny, P. (n.d.). Kernel density classification The naive Bayes classifier Kernel density classification.

John, G. H. G., & Langley, P. (1995). Estimating Continuous Distributions in Bayesian Classifiers. *IN PROCEEDINGS OF THE ELEVENTH CONFERENCE ON UNCERTAINTY IN ARTIFICIAL INTELLIGENCE. Montreal, Quebec, Canada*, *1*, 338–345. http://doi.org/10.1.1.8.3257

Learning Classifiers based on Bayes Rule. (n.d.). Retrieved from www.cs.cmu.edu/~tom/mlbook.html.

Logistic Regression. (n.d.). Retrieved March 24, 2016, from http://www.holehouse.org/mlclass/06_Logistic_Regression.html