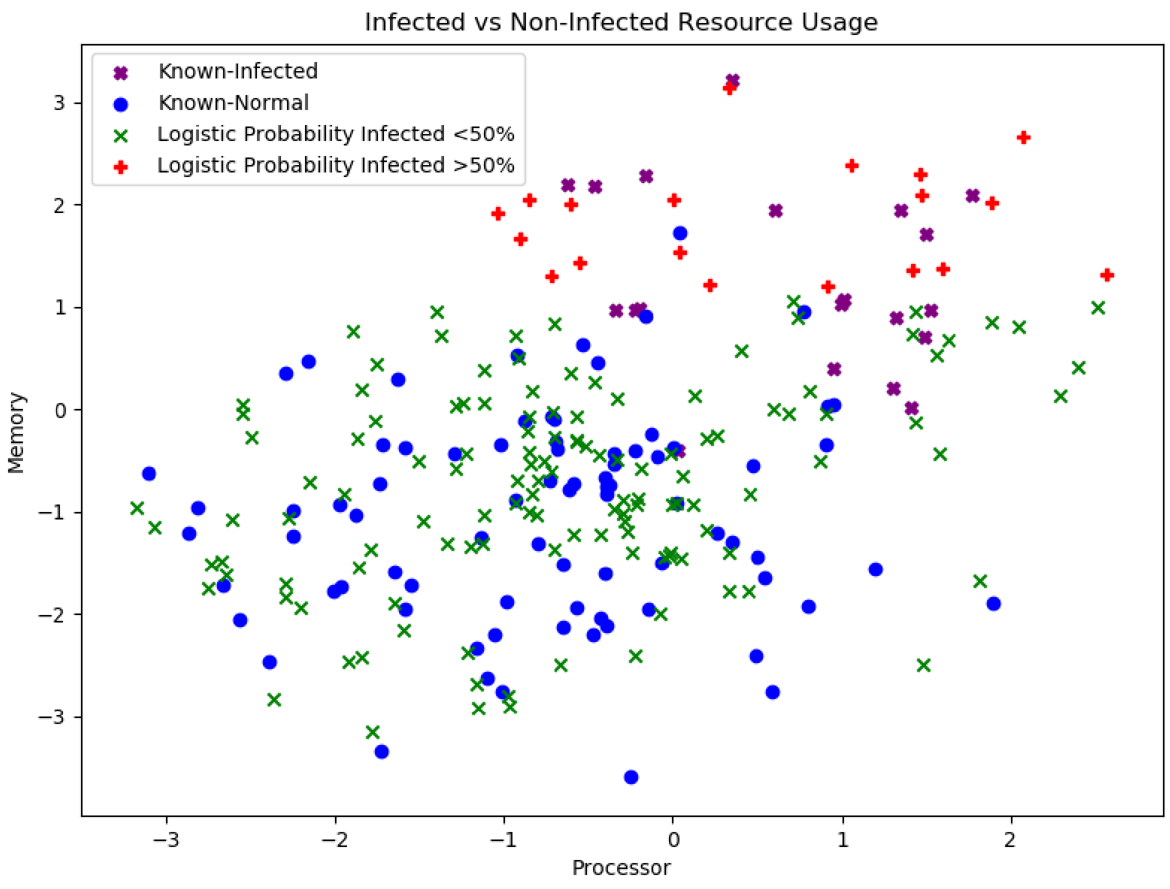
Detecting Infected Hosts Using Machine Learning



Author: Lewis Lyons

Email: lewis_lyons@hotmail.co.uk

Contents

Introduction 2

Data Extraction & Graphing 2

Excel Solution 2

Python Solution 2

Methodology 3

Data Extraction & Graphing 3

Excel Solution 5

Algorithm 5

Prediction 6

Python Solution 7

Logistic Regression Background 7

Algorithm 8

Introduction

During our work we note through manual analysis that several hosts we are responsible for are infected with malware. However, we are responsible for hundreds of hosts and this threat is too pressing to conduct a manual review of each one individually. As such we construct an automated solution which can identify the probability of a host being infected by filtering data about their CPU and memory usage. We test a small sample of hosts manually and create the file “CPU and Memory Data.csv”; we derive all other data from this initial file.

The solution to this problem is constructed of three parts. Part 1 is the data extraction and mapping phase. Part 2 is a binary solution implemented in Excel using a simplistic machine learning algorithm. Part 3 is a probabilistic solution implemented in Python using machine learning based Logistic Regression.

Data Extraction & Graphing

We firstly extracted data from the initial file in various permutations. This allowed us to inspect relationships between the interesting variables and graph the problem. It was decided that a scatter graph depicted the data in the clearest way. To do this we utilised Python. As a language popular for its data manipulating capabilities Python makes the process of data extraction efficient, and through use of functions, repeatable. The graphed data broadly showed a clear positive correlation between increasing processor and memory usage and probability of malware infection on a machine.

Excel Solution

Our Excel solution first defines and trains an algorithm using separately calculated mean values for processor and memory usage in infected and normal systems. Once trained we use the algorithm to make predictions about unknown hosts infection status based on distance from the mean of the average infected machine.

Python Solution

Our Python solution constructs a generalised solution for solving problems of this class and can return probabilities of infection rather than Booleans. The program user can feed in any csv file with the correct data structure through a function. The data is then normalised, and k-fold cross validation is performed to estimate the performance of the learned model on unseen data. We use stochastic gradient decent to minimise the error in our model and logistic regression to generate our predictions. These can then be expressed as probabilities and rounded to 0 or 1 based on the user risk tolerance threshold.

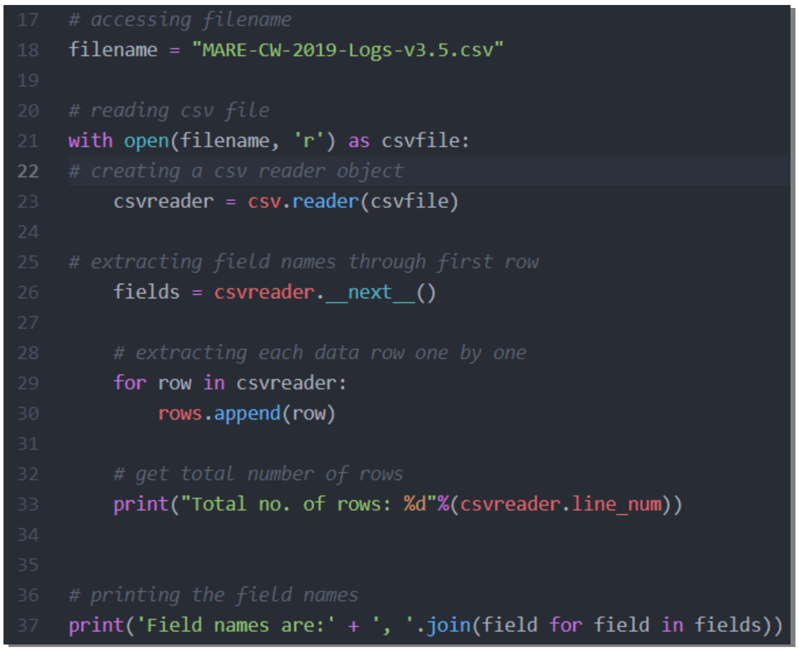
Methodology

This section provides a step by step description and explanation of the work carried out in each part as described in the introduction.

Data Extraction & Graphing

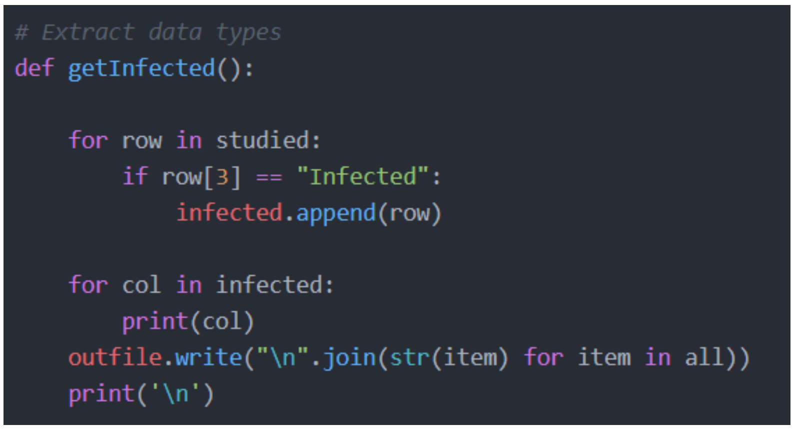
The full code used to extract and graph the data can be found in the 'data_parse_script.py' file.

To access the 'MARE-CW-2019-Logs-v3.5.csv' data we use the inbuilt csv Python module. This allows us to read the csv data using a csvreader object and extract the data row by row.



```python
# accessing filename
filename = "MARE-CW-2019-Logs-v3.5.csv"

# reading csv file
with open(filename, 'r') as csvfile:
# creating a csv reader object
    csvreader = csv.reader(csvfile)

# extracting field names through first row
    fields = csvreader.__next__()

    # extracting each data row one by one
    for row in csvreader:
        rows.append(row)

    # get total number of rows
    print("Total no. of rows: %d"%(csvreader.line_num))


# printing the field names
print('Field names are:' + ', '.join(field for field in fields))
```

We then construct functions which use `for` loops with differing conditional `if` statements to extract permutations of Infected, Normal, and Undefined data sets for later examination.



```
# Extract data types
def getInfected():

    for row in studied:
        if row[3] == "Infected":
            infected.append(row)

    for col in infected:
        print(col)
    outfile.write("\n".join(str(item) for item in all))
    print('\n')
```

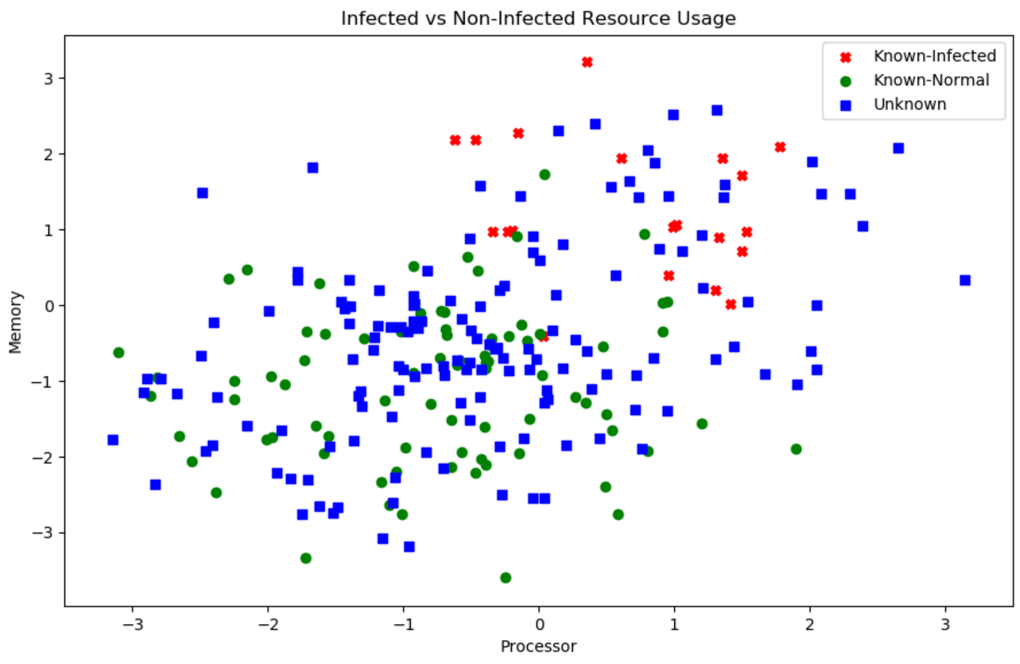
We choose to extract the following sets:

- Infected data only
- Normal data only
- Undefined data only (not known if normal or infected)
- Infected and Normal data combined

With the data extracted we move onto graphing it. The aim of graphing our data was to attempt to notice some inherent pattern which is non obvious in the csv but might be readily available in an appropriate visual form.

We used the Matplotlib module to graph our data in Python. Matplotlib is available from https://matplotlib.org/users/installing.html or as part of the Anaconda distribution.

The graph below represents a plot of Infected, Normal, and Unknown data.



We note that Known-Infected data points are clustered in the top right of the graph and Known-Normal points are clustered more to the bottom left. This implies positive correlation between increasing resource usage and probability of Infection.

Excel Solution

Our objective is to have our algorithm predict, with a reasonable degree of accuracy, if an unknown system in our data set is infected or not. To begin we will define a simple algorithm we can train.

Algorithm

Step 1: Calculate the average processor and memory usage for known-normal machines.

Step 2: Calculate the average processor and memory usage for known-infected machines.

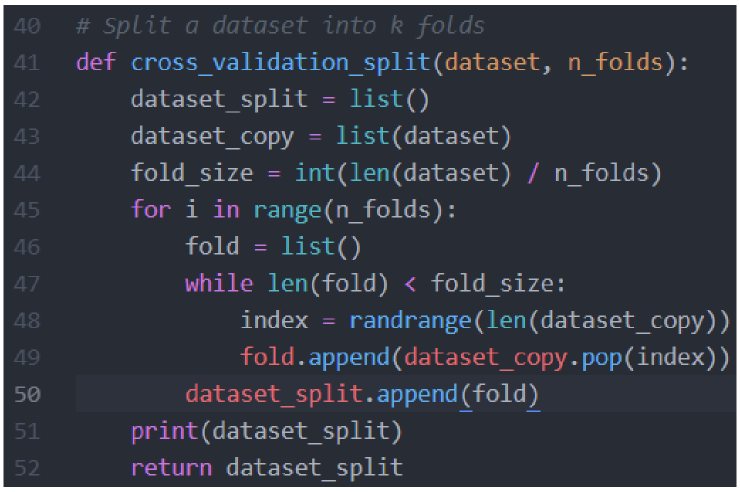
Step 3: If the processor and memory usage are closer to the mean infected machine, designate it as infected.

Step 4: If the processor and memory usage are closer to the mean normal machine, designate it as normal.

Steps 1 & 2 train our algorithm. In this case this is a very simplistic training method, however given how segregated the data appears in the scatter plot it is plausible that simple is all that is needed for reasonable accuracy. Steps 3 and 4 allow us to formulate predictions.

Next, we validate the algorithm. To do this we need to create a way to test how accurately our algorithm makes predictions. It is important to point out that validation should be performed on the data with all variables known. In our case this is the combination of our Infected and Normal sets.

In machine learning validation is performed first by training an algorithm with your data and then testing that algorithm. To do this correctly we need to split our data into training and test sets. In this case splitting our data through simple division would likely be enough however to increase the accuracy of the algorithm we chose to implement cross-validation. Specifically, we used the k-folds method of cross-validation with k=3. The randomising nature of cross-validation meant that we chose to keep our k-value low simply because working with more splits in excel would become extremely cluttered and difficult to understand. The full code to perform the splits can be found in the 'excel_cross_validation.py' file. We use 2/3 of the data for training and 1/3 for testing.



```python
# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    print(dataset_split)
    return dataset_split
```

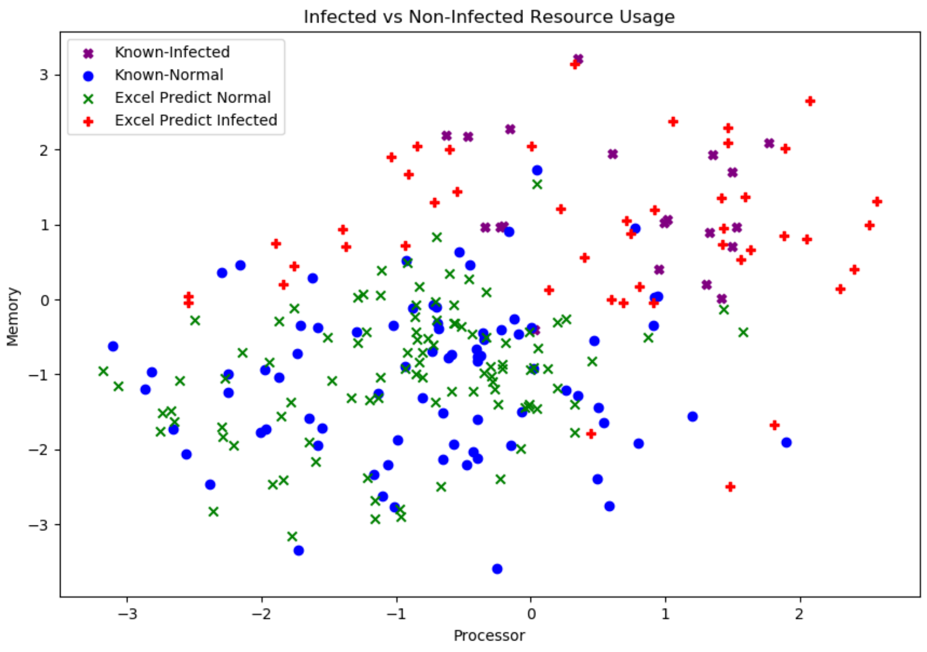
In excel we compute the means for processor and memory usage separately for normal and infected systems. We then use an excel INDEX ARRAY function to compare the processor and memory values of each entry to the infected and normal means of their type and output the closer value. Using these values, we make an initial prediction for processor and memory separately. If the data value is closer to the normal mean we designate it as 0, and if it its closer to the infected mean we choose 1. We then check the accuracy of our basic predictions. This is possible as the training data has known outcomes for infection. A quick evaluation demonstrates that if we simplified our algorithm further and made predictions only when both processor and memory metrics agreed about the state of a machine, we would yield 64% accuracy. To improve this accuracy, we add complexity to allow making a choice in the instance that the means of processor and memory are conflicting with respect to whether a machine is infected. To do this we calculated the difference between the means and their respective independent variables, we then choose the variable with the smallest difference to act as the predictor. This method returned ~84% accuracy during training, a 20% improvement over the previous method.

Having confirmed an acceptable level of accuracy using our training data, we go on to validate our results with our test data. The means are recalculated for this data set and the process is repeated. This test resulted in ~79% accuracy. Given the small sample sizes being worked with a +-5% discrepancy in accuracy is not cause for concern.

Prediction

Finally, we calculate the mean values for all the data with known status and use these as the coefficients for predicting our unknown data. The prediction document can be found under 'excel_predictions_final.csv'. Predictions are calculated with the same methodology as in training with the exception that the true values of infection are unknown and so cannot be cross verified.

Having retrieved values of infection for the unknown machines, we graph our data to check if it appears consistent with what we would expect from our hypothesis of increasing resource usage implies increasing malware infection probability.



From observation we note the graphed results are mostly consistent with our hypothesis. Given our algorithm was only found to be ~80% accurate during testing it is expected that results will contain some outliers which deviate from the hypothesis.

We can now make a conclusion as to which machines are likely to be infected and utilise our time more efficiently to address them.

Python Solution

Logistic Regression Background

Our Python solution implements a different learning solution than in Excel. We choose logistic regression for our method for several reasons. Logistic regression seeks to model the probability of an event occurring depending of the values of the independent variables. With this model we can estimate the probability that an event occurs for a randomly selected observation versus the probability that the event does not occur and ultimately predict the effect of a series of variables on a binary response variable. Given we aim to produce a probability of infection such that we can prioritise assessing systems with respect to their risk, logistic regression suits this purpose. In this case our series of variables are processor and memory usage and our binary response is the state of Infected or Normal.

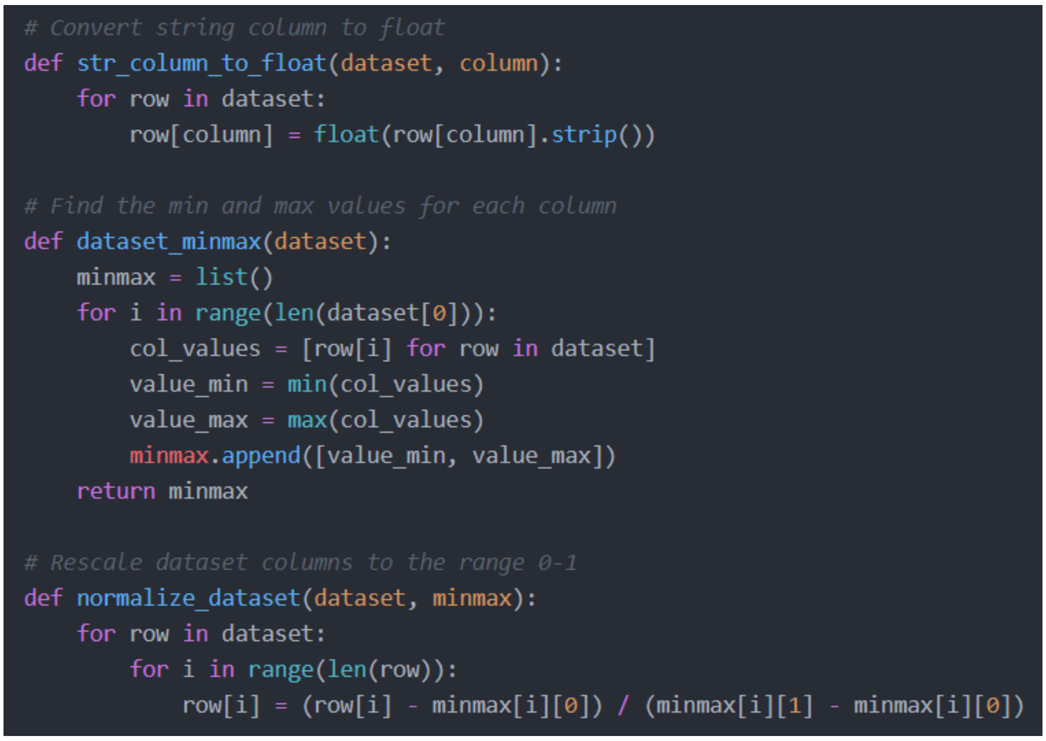
It is worth discussing why other regression methods were not chosen as learning methods for this problem. Simple linear regression cannot be used as it operates on the premise of one quantitative variable predicting another. However, we have a dichotomous dependant variable, Infected/Normal is Boolean not quantitative. Multiple regression is just simple linear regression with more independent variables and so is not viable by the same reasoning. Nonlinear regression uses two quantitative variables, but the data is curvilinear. Most other types of regression require our data to have a normal distribution. As our data has binary components it does not follow a normal distribution.

Algorithm

Preparing Dataset

We load the dataset using the Python csv module. The function `str_column_float()` converts the string values in the data into numeric types and then each column is normalized to values in the range 0 to 1 through the `dataset_minmax()` and `normalize_dataset()` functions.

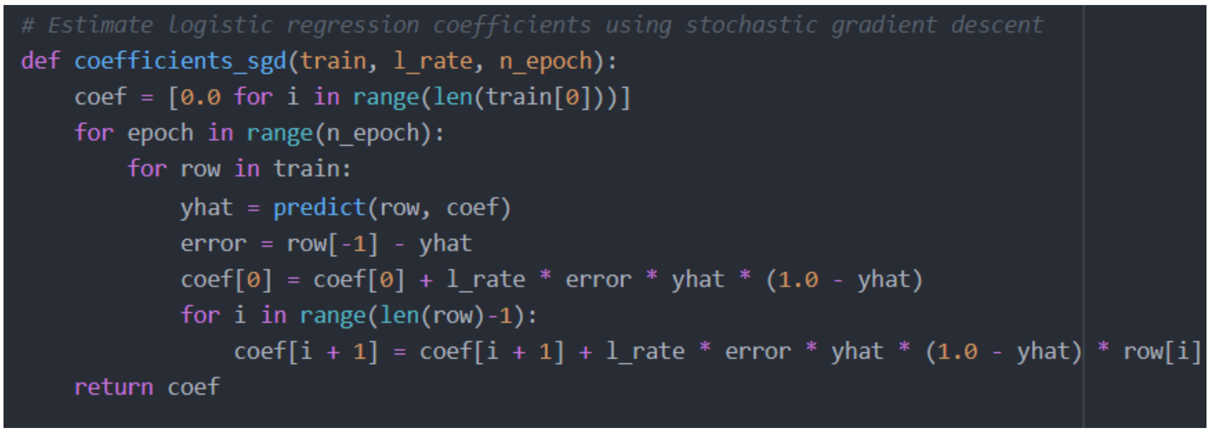
As in the Excel solution we use k-cross validation to estimate the performance of the learned model on unseen data. Each k model will be evaluated using classification accuracy which is the number of correct predictions made divided by the total number of predictions made expressed as a percentage. These processes are accomplished through the `cross_validation_split()`, `accuracy_metric()` and `evaluate_algorithm()` functions.



```python
# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Find the min and max values for each column
def dataset_minmax(dataset):
    minmax = list()
    for i in range(len(dataset[0])):
        col_values = [row[i] for row in dataset]
        value_min = min(col_values)
        value_max = max(col_values)
        minmax.append([value_min, value_max])
    return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
    for row in dataset:
        for i in range(len(row)):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])
```

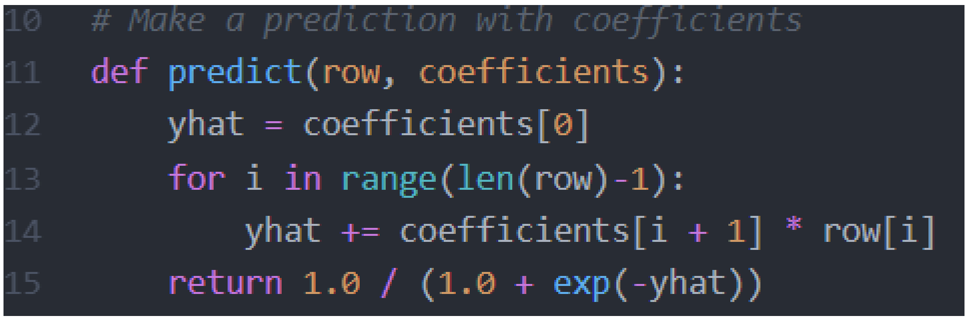
Training Data

We first need to estimate coefficient values for our training data. To do this we will use stochastic gradient decent (SGD). SGD requires a user defined learning rate and a set number of epochs. Through testing we settled on a learning rate of 0.3 and 100 epochs. The coefficients are updated for each row in the training data over every epoch. The degree to which coefficients are adjusted is based on the error the model made. In doing so it begins to learn to predict more accurate answers over time.

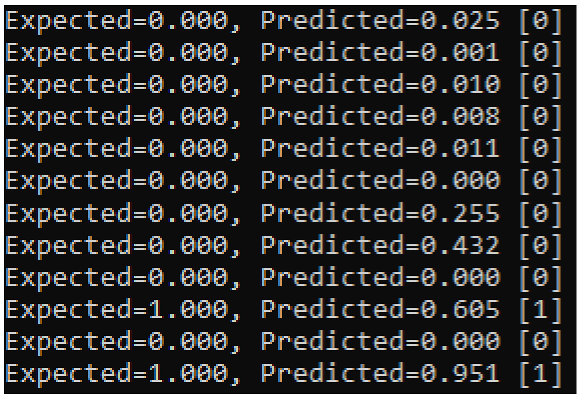


```python
# Estimate logistic regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
    coef = [0.0 for i in range(len(train[0]))]
    for epoch in range(n_epoch):
        for row in train:
            yhat = predict(row, coef)
            error = row[-1] - yhat
            coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
            for i in range(len(row)-1):
                coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
    return coef
```

We then create a prediction function which uses the derived coefficients to make predictions. And test its accuracy. You can run the ‘predict_test.py’ file to repeat this result. Ensure that you comment or uncomment the correct dataframe variable for the file you are looking to test, the default is the training data. The predict function displays a probability and then rounds relative to 0.5.



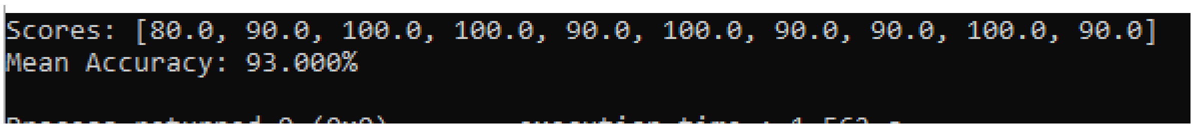
```python
# Make a prediction with coefficients
def predict(row, coefficients):
    yhat = coefficients[0]
    for i in range(len(row)-1):
        yhat += coefficients[i + 1] * row[i]
    return 1.0 / (1.0 + exp(-yhat))
```



```
Expected=0.000, Predicted=0.025 [0]
Expected=0.000, Predicted=0.001 [0]
Expected=0.000, Predicted=0.010 [0]
Expected=0.000, Predicted=0.008 [0]
Expected=0.000, Predicted=0.011 [0]
Expected=0.000, Predicted=0.000 [0]
Expected=0.000, Predicted=0.255 [0]
Expected=0.000, Predicted=0.432 [0]
Expected=0.000, Predicted=0.000 [0]
Expected=1.000, Predicted=0.605 [1]
Expected=0.000, Predicted=0.000 [0]
Expected=1.000, Predicted=0.951 [1]
```

The `logistic_regression()` function uses both `coefficients_sgd()` and `predict()` functions to calculate the logistic regression the data.

Finally we are able to evaluate our model using the `evaluate_algorithm()` function which returns the accuracy of each k_fold and then the mean accuracy of all folds.



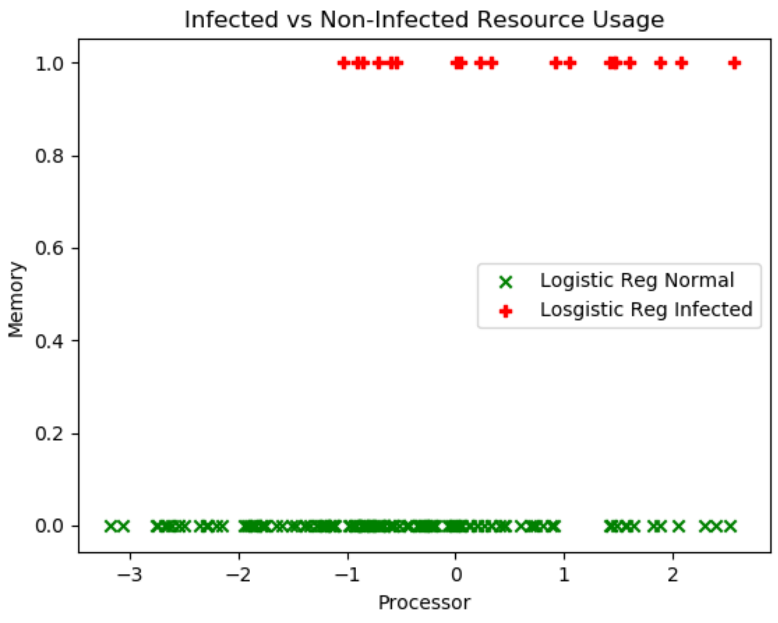
```
Scores: [80.0, 90.0, 100.0, 100.0, 90.0, 100.0, 90.0, 90.0, 100.0, 90.0]
Mean Accuracy: 93.000%
```

As shown above our model was able to make predictions on the test date with 93% accuracy.

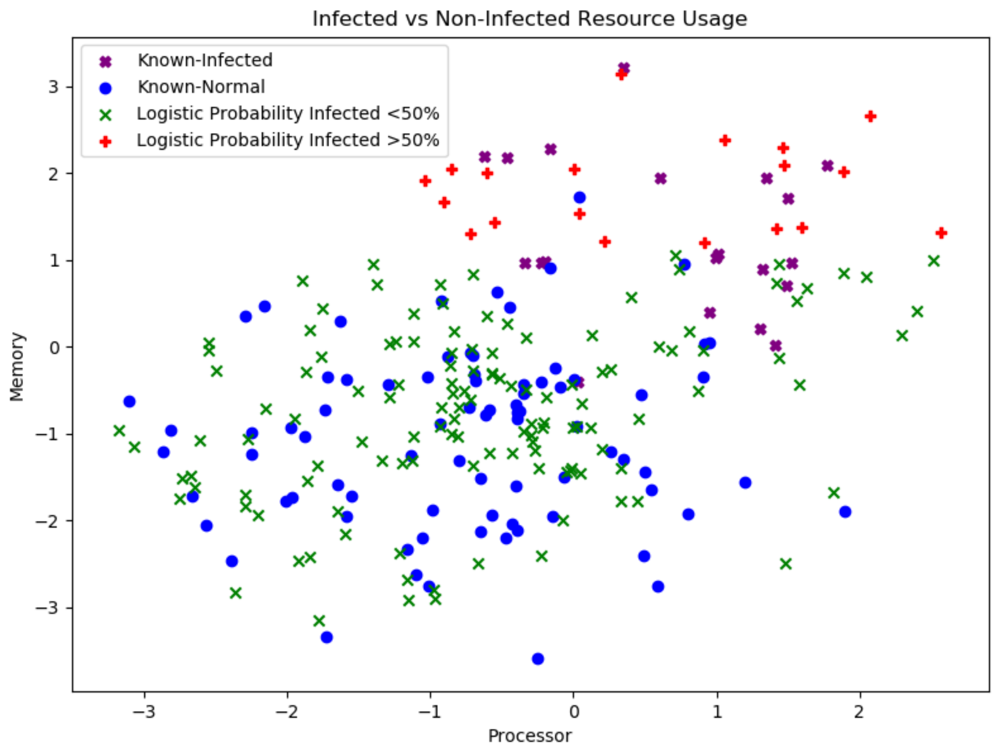
Making Predictions on Unknown Data

It is now simply a case of feeding our unknown data into the `predict()` function and graphing the results.

First, we create a conventional binary logistic regression plot



We then graph our data based on if there is a greater than 50% probability of infection. Importantly we can use a for loop to extract data at whatever probability threshold is acceptable. For example, if we were more risk averse, we could label all machines with >30% probability of infection as infected. The results offer a much clearer distinction than the excel model and more closely conform to our hypothesis.



Infected vs Non-Infected Resource Usage
Known-Infected
Known-Normal
Logistic Probability Infected <50%
Logistic Probability Infected >50%
Memory
Processor