Project: Mobile App

Individual/Pair Assessment worth 35%

Title: Graphic Simulation of a Machine Learning Classifier

Related outcomes from the unit outline:

1. Apply advanced knowledge to analyse and design mobile software
2. Program a simple mobile application.
3. Interpret metrics produced by a relevant framework to build an effective mobile software product.

Due date: 25/10/2019, 09:00am, WST.

Suggested Length: None specified.

Submission instructions:

To be submitted electronically via Blackboard

Referencing:

As per ECU's APA guidelines

Academic Misconduct:

Please ensure that you have read ECU's policy on academic misconduct

Task list:

- Implement an app using the Corona SDK and Lua that demonstrates the principle of the kNN algorithm. Pre-clustered data will be supplied.
- Create a three-minute video, demonstrating the functionality of your app
- Create a document containing:
  - A list of the functional requirements for the app
  - A diagram of the app screens (you might like to use a navigation map as you used in Systems Analysis)
  - Evidence of appropriate configuration management and test-driven development
  - A record of the black box and unit tests you have completed to verify that your app works as expected
  - A COCOMO predictive model, based on your pooled data in your team (include three PSP time logs per person)

Background:

Testing:
Black box tests are based on the requirements (so you should list the requirements) and test end-to-end functionality. In that sense, if you can define the input and output of some aspect of functionality, then you can design a black-box test to confirm that

the function works. In contrast, unit testing involves testing an individual software component/module/function and may also require developing test driver modules or test harnesses. Select a suitable unit testing framework (LuaUnit is popular, but not the only choice). Not all Lua frameworks integrate with Corona. This may not be an issues if you partition your code and separate functionality from UI constructs. Test-driven development means writing unit tests before functional code as part of the build process. The unit test is based on one or more requirements. You then write code that passes the test (therefore meeting the requirement).

Machine Learning:
Machine Learning (ML) is proving to be a popular set of techniques, especially in cyber security. In ML, we usually wish to either predict some future value or classify some input. One simple algorithm for classification is k-Nearest-Neighbour (kNN). If we have a dataset of known classes (called the training set), this is known as supervised learning because we know the answer (the classes) in advance. We can then present a new value and use kNN to determine to which class this new value belongs. This assignment is about implementing and testing an app to simulate KNN, showing how it works and allowing a user to input a target value to determine its class.

kNN:
The k factor in kNN is the number of neighbours used to determine the class of an input. If we graphed k vs. the error rate (mis-classifications), the graph would be non-linear i.e., for a given problem there is an optimal k (or range of k values), so it would not necessarily be true that larger k means a better classifier. kNN works by calculating the distance between each of the training set values and the target (unknown class) input, using some measure of distance. It then keeps the closest k values and uses those to determine the class of the target (by simply calculating which class has the most instances in the k set).

Distance metrics:
The most common measure is Euclidean distance. This is the straight-line distance between two points. For two dimensions, the formula is: $\mathrm{sqrt}((x_2 - x_1)^2 + (y_2 - y_1)^2)$.

Another measure is the Manhattan distance, where travel is only allowed in the direction of the axes. For two dimensions, the formula is: $((x_2 - x_1) + (y_2 - y_1))$.

The above measures are special cases of the Minkowski distance.

Another measure is the Cosine similarity. There are several other choices.

A problem with simple voting to determine a class is that we gave equal weight to the contribution of all neighbours in the voting process. This may not be realistic as the further away a neighbour is, the more it deviates from the real result. We should be able to trust the closest neighbours more than the farther ones. We can assign weights to the neighbours using the harmonic series. In this scheme, the nearest neighbour is assigned a weight 1/1 (i.e., 1), the second closest is assigned a weight of 1/2 and then continuing up to 1/k for the furthest neighbour.

PSP:
The Personal Software Process (PSP), as espoused by Watts Humphrey, attempts to focus on improving individual performance in the upper levels of the CMM (the Capability Maturity Model). The PSP recommends establishing metrics to measure

aspects of the software process and to test personal competency by writing programs measured with these metrics. Humphrey (1996) contains some background on the PSP.

Each person is required to create three modules/functions in Lua. The simplest way to do this is to select three different-sized parts of the app over the lifetime of the project. Keep accurate records of three things: 1) the size of the code (in lines, not counting comments); 2) how long it takes to write the module/function (include testing time); and 3) any defects (bugs) found and fixed as part of the development.

Record any defects found on the time recording log (refer to Appendix B for defect types).

COCOMO:
Software Engineering is about managing process. By managing process, we obtain a product that has consistent quality attributes. Quality thus transfers to the next project. Therefore, software engineers spend time and effort measuring and recording details about the production of software, especially trying to predict future behaviour from past data. Many predictive models for software estimation, such as COCOMO or the Constructive Cost Model (Boehm, 1981), are of the general form:

Person-Months = a*KLOC^b

where "a" and "b" are defined constants, "*" and "^" are the usual multiplication/exponentiation operators and KLOC means thousands of lines of code (in your model, you would replace person-months with person-days or hours and KLOC with LOC as your programs are small).

When using this type of model, two questions arise. First, where do these constants come from and second, could this model be used as-is or would it need to be calibrated for each organisation or application domain?

The constants come from fitting a curve (the above equation) to known data about projects (their KLOC and PM results). Calibration is essential as the type of projects that were used to derive the original values for the modes of COCOMO are not necessarily in the same domain as the software development firm which wishes to use the model.

A common method used to fit curves to data is least-squares regression. For a linear model, the equation is often expressed as: y = m*x + b, where "m" is the slope of the line and "b" is the point where the line crosses the y-axis (called the y-intercept).

Calculate constants for a predictive model, based on your pooled data. There should be six data points in each team of two people. Use the last four LOC/time data points for this purpose. Keep the first two points (i.e., the data from the first module/function written by each person) aside for validation. The COCOMO equation will need to be transformed into a linear equation to use the linear model above. Use: Log Person-days (or hours or minutes) = Log a + b * Log LOC (in this case "Log a" represents the "b" parameter in y = m*x + b.

There are several tools/packages/languages you could use to perform this task (e.g., Excel, R, SPSS, Python).

Having created a model that predicts time from LOC, use the actual values for LOC from the first two Lua modules/functions written to predict the time to complete the first two Lua tasks. Are these estimates similar to the actual time taken for each? Discuss the difference or similarity.

References

Boehm, B. (1981). Software Engineering Economics. Prentice-Hall.

Humphrey, W.S. (1996). Using a Defined and Measured Personal Software Process. *IEEE Software*. (13)3. May, 77-88. DOI=http://dx.doi.org/10.1109/52.493023

Appendix A - PSP Time Recording Log

Student:

Module/Function Name:

Code Location:

Start Date and Time	Stop Date and Time	Delta Time	Comments

Appendix B – PSP Defect Types

Type Number	Type Name	Description
10	Documentation	Comments, messages
20	Syntax	Spelling, punctuation, typos, instruction formats
30	Build, Package	Change management, library, version control
40	Assignment	Declaration, duplicate names, scope, limits
50	Interface	Procedure calls and references, I/O, user formats
60	Checking	Error messages, inadequate checks
70	Data	Structure, content
80	Function	Logic, pointers, loops, recursion, computation, function defects
90	System	Configuration, timing, memory
100	Environment	Design, compile, test, or other support system problems