Software Cost Estimation

ATSE Assignment 4 - Vlady Veselinov

Contents:

1. Overview of Datasets

Three datasets were selected from the promise repository: Nasa, Kemerer and Use Case Point. Before discussing them, some important aspects about data preprocessing need to be made first. To prepare for analysis, the data needs to contain meaningful values. If the problem being solved doesn't objectively benefit from having a specific column in, it's a good idea to remove that column from the creation of the model. In other words, **preprocessing** is the activity of removing irrelevant information from the dataset and / or reducing the data to actionable features. In order to figure out what these use features are, the **data needs to be explored first**. A good tool to do that is Weka, it allows to build models predicting a column from the dataset and displays each variable that goes into it along with a weight. The weights can be a good indication of the meaning of each variable. For example, in cost estimation time spent is usually a very good indicator about how much the effort/cost will be. **Categorical variables** can be very useful, but they increase the dimensionality of the data. To be usable, they need to be converted to numbers. If there are any arbitrary values such as IDs they need to be ignored during the exploration of the data. Numeric values on the other hand are very actionable. They can be used to find relationships that lead to an accurate model. IDs are unusable, unless required by project.

a. Kemerer Dataset

Starting with a very small dataset with no description, some exploratory analysis can be made with Weka to establish relationships between values. Linear regression came up with the following model:

```
# === Weka Cross-validation ===
# Attributes:    7
#              Language
#              Hardware
#              Duration
#              KSLOC
#              AdjFP
#              RAWFP
#              EffortMM
# Test mode:    10-fold cross-validation

# Linear Regression Model:
EffortMM = 53.4674 * Hardware + 0.389  * AdjFP + -294.1583

# === Summary ===
# Correlation coefficient    0.3664
# Root mean squared error    281.0625 <---------- Baseline
```

Looking at it, it seems that only Hardware and AdjFP matter for determining effort.

b. Nasa Dataset

The largest dataset out of the three. It contains more columns and almost ten times more rows than the Kemerer dataset. Exploratory analysis results in the following linear regression model:

```
'''
=== Weka Exploratory Analysis  ===
Linear Regression Model

effort =

  1645.381  * rely=l,vh +
 -2154.8882 * rely=vh +
   780.4065 * data=l,n,h +
 -1748.3515 * cplx=l,vh,xh +
  2064.9888 * cplx=vh,xh +
   396.2673 * time=vh,h,xh +
  -942.7451 * time=h,xh +
  2279.4264 * time=xh +
 -1181.6677 * pcap=vh,h +
   696.6285 * pcap=h +
   781.9602 * apex=l,vh,h +
  -617.567  * apex=h +
  1010.5671 * ltex=n,h +
  -704.447  * ltex=h +
  -474.0185 * sced=h,n +
      6.2114 * kloc +
     -0.1954 * defects +
     88.5857 * months +
 -2166.5075


=== 10-fold Cross-validation ===

Correlation coefficient                  0.6421
Root mean squared error               1085.1159
'''
```

This time there are many more variables in play, most of them from the categories, i.e. non-numerical types of data. From the numerical columns, kloc and months seem to be the most important.

c. UCP Dataset

This dataset describes the development of software, with methods and technologies used. It contains two effort columns, one scaled by a productivity factor, it is removed in order to keep things simple. The column value that is to be predicted is “Real_Effort_Person_Hours”. Other unusable columns that are removed before analysis: ID and data donator.

Looking back at the other datasets, there were some problems. The first one (Kemerer) is too short. The analysis approach the project initially had towards them is, because categorical variables weren’t used at all, only numerical. This was a problem because they can contain valuable information. Weka analysis was also suffering from not offering more than one comparison baseline. To correct that, the Gaussian Processes baseline is introduced. Here’s how Weka analysis looks for the UCP dataset:

```
# === Weka Exploration of Data ===
# Attributes:    15
#                Simple Actors
#                Average Actors
#                Complex Actors
#                UAW
#                Simple UC
#                Average UC
#                Complex UC
#                UUCW
#                TCF
#                ECF
#                Real_Effort_Person_Hours
#                Sector
#                Language
#                Methodology
#                ApplicationType
# Test mode:     10-fold cross-validation

# === Linear Regression Model ===

# Real_Effort_Person_Hours =

#    -340.4336 * Complex Actors +
#      45.4729 * Complex UC +
#     304.2088 * Sector=
#          Service Industry,
#          Banking,Professional Services,
#          Wholesale & Retail,
#          Electronics & Computers,
#          Communication,Manufacturing +

#    -880.3201 * Sector=Communication,Manufacturing +
#    1152.557  * Sector=Manufacturing +
#    ... and many more ...
#     424.522  * Language=C#,Oracle,C,Java,XML,C++,.Net,CSP +
#    1161.8863 * Language=Oracle,C,Java,XML,C++,.Net,CSP +
#   -1065.6929 * Language=C,Java,XML,C++,.Net,CSP +
#    1223.438  * Language=CSP +
#   -1187.4421 * Methodology=
#    ... and many more ...
#    2117.1876 * ApplicationType=Real-Time application +
#    7045.5009

# Correlation coefficient                  0.1599
# Root mean squared error    860.7491 <------- Linear Regression Baseline



# === Gaussian Processes ===
# Correlation coefficient    0.3372
# Root mean squared error    658.8125 <------- Gaussian Processes Baseline
```

This further clarifies that categorical columns are important and the exploratory model is too complex for anyone to make conclusions with a single glance about what variables are most important. This means that the GA model needs to consider those values somehow, they can't be discarded. They need to be accounted for preferably in a numerical form. A solution to this is to create a column for every category-value pair that will contain a 0 or a 1. For example: "language_c++: 0" shows that for a specific row, c++ wasn't used. These types of columns are called dummies in the Pandas Python library, an example of making these in practice will be shown in the implementation section further down.

2. Implementation

a. Genetic Programming in Deap

The Deap Python library contains an excellent set of tools for genetic programming problems. A genetic algorithm will generate a binary tree, that is the individual in the population. The binary tree can be formed from a set of primitive functions, i.e. a primitive set:

```python
import operator, math, random, numpy, os
from deap import base, creator, gp, tools, algorithms

# Create primitive set with name and arity (how many arguments it can take)
pset = gp.PrimitiveSet('EFFORT', 2)
pset.addPrimitive(operator.add, 2)
```

This works well when functions can't throw exceptions, but what if they can? Division by 0 for example. Situations like these can be cheated by adding safe primitive functions:

```python
# Prevent division by zero
def div(x, y):
    try:
        return x / y
     except ZeroDivisionError:
        return 1

pset.addPrimitive(div, 2)
```

How about generating random constants?

```
# Float constant between 0.01 and 10
pset.addEphemeralConstant('floats', lambda: random.uniform(0.01, 10))
```

From there on, a simple genetic algorithm toolbox is created using the primitive set:

```
# Aim to minimise fitness
creator.create('FitnessMin', base.Fitness, weights=(-1.0,))
creator.create(
    'Individual',
    gp.PrimitiveTree,
    fitness=creator.FitnessMin,
    pset=pset
)

toolbox = base.Toolbox()
toolbox.register(
    'expr',
    gp.genHalfAndHalf,
    pset=pset,
    min_=1,
    max_=2
)

toolbox.register(
    'individual',
    tools.initIterate,
    creator.Individual,
    toolbox.expr
)

toolbox.register('population', tools.initRepeat, list, toolbox.individual)
toolbox.register('compile', gp.compile, pset = pset)

toolbox.register('select', tools.selTournament, tournsize = 3)
toolbox.register('mate', gp.cxOnePoint)
toolbox.register('expr_mut', gp.genFull, min_ = 0, max_ = 2)
toolbox.register('mutate', gp.mutUniform, expr = toolbox.expr_mut, pset =
pset)
```

Note that a fitness function is missing, it will be registered when the toolbox is used during cross-validation.

b. Preprocessing data with Pandas

To revisit the overview, the preprocessing steps that need to be taken are:

- Read the file
- Remove irrelevant columns
- Fill any gaps in the data (null values)
- Add dummy columns for each category-value

All of this can be conveniently done in 4 rows of code:

```
import pandas as pd

df = pd.read_csv(data_path, sep=';')

# Drop irrelevant columns
df = df.drop(['Project_No', 'DataDonator', 'Real_P20'])

# Fill empty values with zeros
df = df.fillna(0)

# replace categorical columns with dummy columns with binary numbers
# i.e. replace column 'Sector' with 'Sector_value1', 'Sector_value2'...
df = pd.get_dummies(df, columns=['Sector', 'Language', 'Methodology'])
```

c. K-Fold Cross-Validation

To prove that the generated model is performing well, cross-validation needs to be used. Basically the data is split into two train and test parts so each time the model is tested with data it hasn't seen yet. This can be done K amount of times, hence "K-Fold". More here.

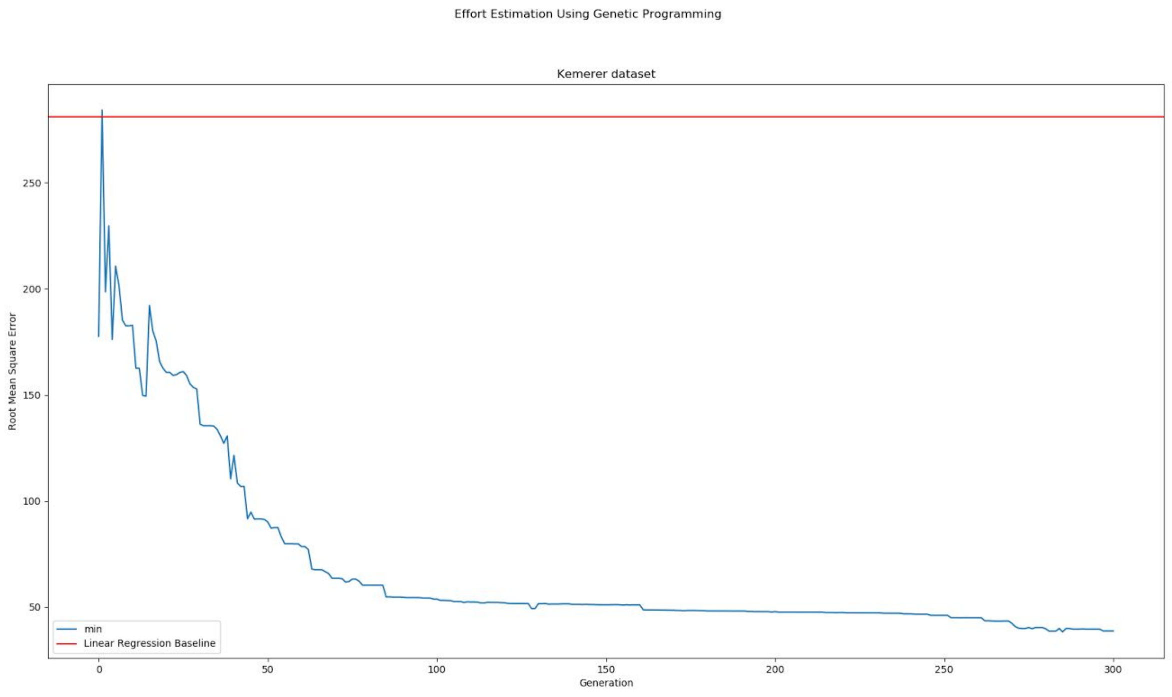
```
from sklearn.model_selection import KFold

kf = KFold(n_splits = 10, shuffle = True)
for trainingIndices, testingIndices in kf.split(df):
    # Train GA
    # Test GA
```

3. Results

a. Kemerer

The first datasets were seemingly performing well, but there wasn't any cross-validation. For example, here is the small Kemerer dataset. Blue is the GA, args are in column order:

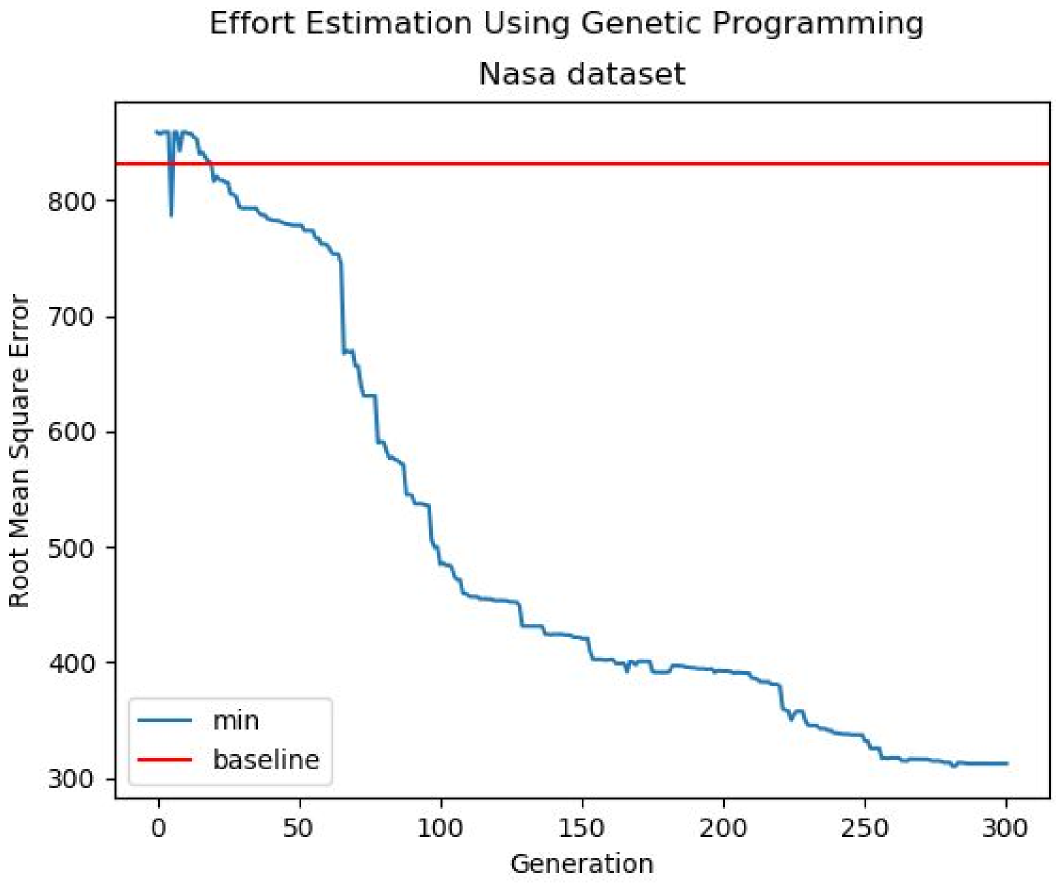


In the end it came up with a model like this:

```
mul(sqrt(add(mul(sqrt(add(add(sqrt(7.667999712141599), add(sqrt(ARG1),
mul(sqrt(div(sqrt(add(ARG0, ARG1)), ARG1)), add(div(ARG1,
add(sqrt(2.0194298119934584), div(sqrt(ARG1), sub(9.69260900025867,
add(sqrt(sqrt(ARG0)), sqrt(ARG1)))))), add(ARG0, ARG0))))),
mul(sqrt(add(ARG1, sqrt(sqrt(ARG0)))), ARG1))), div(ARG1, add(ARG0,
div(sqrt(ARG1), sub(9.69260900025867, mul(ARG0,
add(sqrt(sqrt(add(sqrt(ARG1), add(mul(ARG0, mul(ARG0, 7.667999712141599)),
ARG1)))), ARG0))))))), add(7.667999712141599,
mul(sqrt(add(mul(sqrt(sqrt(add(sqrt(sqrt(sqrt(ARG0))), ARG0))),
sub(div(ARG1, add(sqrt(sqrt(7.667999712141599)), div(sqrt(ARG1),
sub(9.69260900025867, add(sqrt(sqrt(ARG0)), mul(ARG0, mul(ARG0,
7.667999712141599))))))), sub(add(ARG1, sqrt(2.0194298119934584)),
add(sqrt(ARG1), div(div(mul(7.667999712141599, 7.667999712141599),
sub(div(div(ARG1, 9.012357525628882), sqrt(ARG1)), add(sqrt(ARG0),
sqrt(add(ARG0, 5.458368815782998))))), sqrt(sqrt(ARG0))))))),
add(add(add(7.667999712141599, 8.421383262353041), ARG0), add(sqrt(mul(ARG0,
mul(ARG0, 7.667999712141599))), div(sqrt(mul(sqrt(add(ARG1, add(sqrt(ARG1),
add(ARG1, ARG1)))), ARG1)), sub(mul(ARG0,
add(sqrt(sqrt(add(sqrt(7.667999712141599), add(ARG1, ARG1)))),
sin(sqrt(7.667999712141599)))), 7.667999712141599)))))), div(ARG1,
ARG0))))), log10(add(ARG0, sqrt(sqrt(add(ARG0, ARG1))))))
```

b. Nasa

The Nasa dataset analysis suffered from the same lack of cross-validation, so this performance doesn't truthfully speak about the GA's ability to generate a nice model.

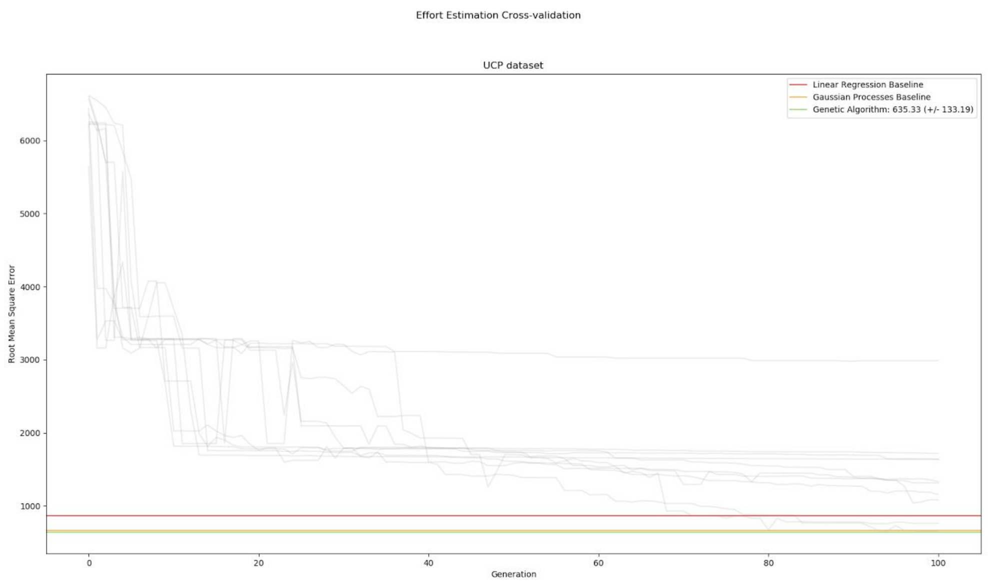


The best model from the evolution:

```
mul(sub(ARG1, 6.0425291304844135), sqrt(add(add(div(ARG1, cos(sqrt(mul(ARG1,
sqrt(sub(cos(div(ARG0, ARG1)), add(cos(ARG1), sub(sub(ARG0,
add(add(6.0425291304844135, ARG0), ARG1)), 1.828653421186695)))))))),
add(div(ARG1, cos(sqrt(sub(add(ARG1, 6.0425291304844135), add(ARG0,
sub(ARG0, add(sub(ARG0, add(add(ARG1, ARG1), ARG1)), add(ARG1, div(ARG1,
cos(sub(ARG0, add(ARG1, ARG1)))))))))))), sub(mul(ARG1, ARG1),
sub(cos(add(log10(add(1.68312489879983, ARG0)), 0.9044975227240548)),
1.828653421186695)))), sub(add(ARG0,
sub(add(add(add(cos(2.9295970723178715), ARG0), mul(sub(add(ARG1, ARG1),
sub(sub(ARG0, add(ARG1, ARG1)), 1.828653421186695)), 6.0425291304844135)),
add(add(sub(ARG0, add(ARG1, sub(cos(ARG1), add(cos(ARG1), sub(sub(ARG0,
add(add(6.0425291304844135, ARG0), ARG1)), 1.828653421186695))))), add(ARG1,
div(ARG1, cos(sub(ARG0, add(ARG1, ARG1)))))), ARG1)), add(ARG1, ARG1))),
div(add(mul(ARG1, sqrt(sub(add(sqrt(sub(ARG1, add(div(ARG1, cos(sub(ARG0,
add(ARG1, ARG1)))), ARG1))), 6.0425291304844135), add(ARG0, sub(ARG0,
add(ARG1, ARG1)))))), sub(sin(add(add(add(add(add(add(add(ARG1, ARG1),
sqrt(ARG1)), ARG0), div(mul(2.79977835089251, ARG0), add(4.211402125706582,
ARG0))), ARG0), mul(sub(add(ARG1, ARG1), div(add(add(ARG1, ARG1), add(ARG1,
ARG1)), ARG0)), ARG0)), add(cos(ARG1), mul(cos(ARG0), div(4.843388281310775,
sub(sub(ARG0, ARG0), 1.828653421186695)))))), add(ARG1, add(ARG1, ARG1))),
add(cos(ARG1), 0.9044975227240548))))))
```

c. UCP

The analysis of this dataset takes care of all the above errors by actually using cross-validation and comparison to more than one baseline… See image with higher resolution.



Each of the grey lines represents fold of the GA being trained and evolved. There are certainly instances that (this time) objectively outperform the Weka exploratory baselines. Here’s an example of a good individual:

```
add(add(add(sub(sub(add(add(sub(sub(add(
sub(mul(9.629349975112635, 9.629349975112635), ARG7),
mul(mul(9.629349975112635, 9.629349975112635), 9.629349975112635)), ARG5), ARG7),
add(sin(add(ARG5, mul(9.629349975112635, 9.629349975112635))),
mul(mul(9.629349975112635, 9.629349975112635), 9.629349975112635))),
add(div(log10(sub(9.629349975112635, ARG7)), mul(mul(9.629349975112635,
9.629349975112635), 9.629349975112635)), 9.629349975112635)), ARG5), ARG7),
add(mul(9.629349975112635, 9.629349975112635),
add(sub(sub(add(sub(mul(9.629349975112635, 9.629349975112635), ARG7),
mul(mul(9.629349975112635, 9.629349975112635), 9.629349975112635)), ARG5), ARG7),
add(sub(sub(add(sub(div(log10(sub(ARG7, ARG7)),
add(add(sub(ARG7, ARG7), add(9.629349975112635, ARG44)),
mul(mul(mul(9.629349975112635, 9.629349975112635), 9.629349975112635),
9.629349975112635))), ARG14), mul(mul(9.629349975112635, 9.629349975112635),
9.629349975112635)), ARG5), ARG7), add(div(ARG33, mul(sub(add(9.629349975112635, ARG5),
9.629349975112635), ARG30)), add(sub(sub(add(sub(mul(sub(ARG42, ARG0),
9.629349975112635), ARG7), mul(mul(9.629349975112635, 9.629349975112635),
9.629349975112635)), 9.629349975112635), ARG7), add(9.629349975112635,
9.629349975112635))))))), add(div(mul(9.629349975112635, 9.629349975112635),
add(div(log10(9.629349975112635), ARG11), 9.629349975112635)),
sub(sub(add(sub(div(div(log10(9.629349975112635), ARG11),
add(ARG7, mul(9.629349975112635, 9.629349975112635))), ARG14),
mul(mul(9.629349975112635, 9.629349975112635), 9.629349975112635)), ARG5), ARG7))),
mul(9.629349975112635, add(sub(ARG7, ARG5),     mul(9.629349975112635,
9.629349975112635))))
```

This indicates how the evolutionary method is good at handling functions with high arities. In all cases the arguments are ordered in the same sequence as the dataframe columns.

To summarise, this project should have properly considered how to use categorical columns from datasets and how to implement cross-validation. Genetic programming certainly seems to perform well compared to common machine-learning techniques such as linear regression and gaussian processes. As an improvement to this, it would be nice to implement tree simplification at some point, in order to avoid duplicate operators because they make the functions unreadable. It’s unclear how simplifying the tree in-between generations would affect the improvement rate, but it could certainly help human readability to do it when the GA is done. Please try running the example repo, instructions are in the readme.