I. GENETIC PROGRAMMING

A. Introduction

We use a supervised ML algorithm called genetic programming(GP)[?]. A detailed introduction to this method is beyond the scope of the paper, so we only present essential information pertaining to our work.

GP is evolutionary computation that employs naturally occurring genetic operations, a fitness function, and multiple generations of Darwinian evolution to resolve a user-defined task [?]. GP can be used to discover a mathematical relationship between the input variables, also called features, in data (regression) or group data into categories (classification). Similar to biological natural selection, GP learns how well the program functions by comparing the output of its multivariate expression to a *fitness score*. Programs with high fitness score are most likely, but not certain, to propagate to next generation. With subsequent generations, programs are more likely to get better at solving the task at hand [?].

GP multivariate expressions are classically represented as a syntax tree, where the trees have a root (top center), nodes (mathematical operators), and leaves (operands). Operators can be arithmetic, trigonometric, boolean, etc. and the operands are place-holder variables related to the problem. The tree depth can be varied aiding in evolution of more complex multivariate expression. GP intitially generates a stochastic poplulation of trees, computes fitness scores and randomly selects trees for comparison. The lead scoring trees are selected and one of the genetic operators (reproduction, mutation, crossover) is randomly applied to carry them forward to next generation. The process repeats until user-defined conditions are met. The run-time parameters like intial population size, tree depth, tournament size for competing trees, number of generations, and termination criterion can be tuned for better algorithmic performance.

The unique aspect of GP algorithm used here is the transparency of its internal workings. Unlike many black box ML algorithms, the evolutionary process can be reviewed at each step which might be quite important to many researchers.[?]

For our analysis, we used an open source python code, Karoo GP [?]. Karoo GP package is scalable, with multicore and GPU support enabled by the TensorFlow library and has capacity to work with very large datasets. The latest version of Karoo GP can be found in PyPI [?].

B. Methodology

After the dataset creation ...

The end of each Karoo GP training produces a multivariate exression with heighest fitness score. We repeat the training process 200 times to get 200 expressions for each label (hasNS and hasRemnant). Due to the stochastic nature of the GP algorithm, the expressions mostly tend to be unique depending upon the complexity of the classification problem. Each expression can then be used to classify the reconstructed source parameters to determine the labels hasNS and hasRemnant by substituting the values in the expression. If the result is greater or equal to zero, we classify the reconsturcted set of prameter as true and false otherwise. The performace of each of the expressions against the testing set for one EoS is shown in Fig 1. As seen in the plots, the average performace of hasRemnant expressions is better than hasNS expressions. Also the winning expressions are more dispersed in case of hasRemnant. Shall we explain the reasons: hasNS being simpler classification but contaminated by poor reconstructions by the pipeline. HasRemnant is more complex leading to more variable winning trees. **Calculation of Probability:** Using testing set, we compute probabilities using bayesian method. HasNS and hasRemnant are not inherintly independent quantities since there can be no hasRemnant without hasNS. Hence, we utilize both hasNS winning expressions and hasRemnant winning expressions for quantifying probabilities $p(hasNS)$ and $p(hasRemnant)$. If XREM denotes the number of hasRemnant winning trees which classifies given set of parameter as true and XNS denotes the number of hasNS winning trees which classifies as true, we compute the probability on the testing set as:

$$p(HasNS|XREM \cap XNS) = \frac{p(XREM \cap XNS|HasNS).p(HasNS)}{p(XREM \cap XNS)} \tag{1}$$

$$p(HasRemnant|XREM \cap XNS) = \frac{p(XREM \cap XNS|HasRemnant).p(HasRemnant)}{p(XREM \cap XNS)} \tag{2}$$

There as some caveats to this calculation. Given a finite testing set, we cannot fully compute the entire parameter space (201 x 201 cases) of the probability. Also there are some cases which are highly unlikely to occur, like $p(hasNS|XNS = 0 \cap XREM = 200)$. Also, our probability computations for a likely case might also suffer due to the finite testing set size. Hence, in order to span the entire parameter space of the probability, we use gasussian process regression.

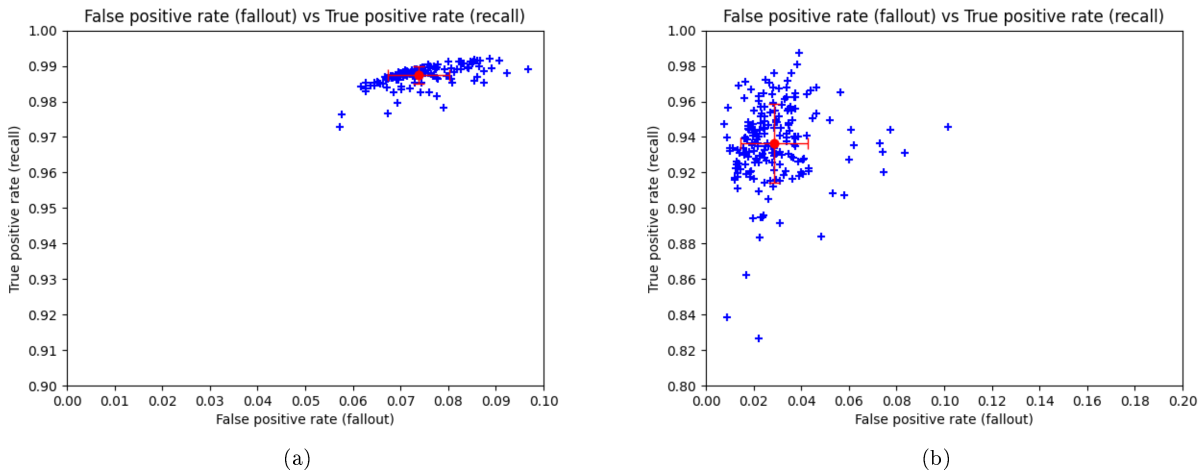

FIG. 1: The plot on the left is a FPR (False Positive Rate) vs (True Positive Rate) plot for all 200 winning expressions for hasNS classification on MS1-PP EoS. The red dot is the average of TPR and FPR of the expressions and the vetricle and horizontal spread are the one sigma values for TPR and FPR respectively. The plot on the right is similar but for hasRemnant classification.