TDE 3 - Genetic Algorithm for Feature Selection

Matheus Girardi, Andrei Silva, Evandro Diniz

This project aimed to make use of a genetic algorithm to generate the ideal subset of features from the breast cancer Wisconsin dataset.

Representing Solutions

Each solution is a `Solution` class (see Code Cell 1), the gene attribute holds a a binary string which encodes the presence of features for that solution, one means the feature is present and zero that it is not.

```python
@dataclass
class Solution:
    gene: str
    fitness: float | None = None
```

Fitness Calculation

In order to calculate to calculate the fitness of a given gene we use a K-Nearest Neighbours (KNN) classifier. We fit it with train data and compute the accuracy over the test set.

```python
    def _fitness_func(self, gene: str) -> float:
        features = [
            self.x_train.iloc[:, i].name for i in range(len(gene)) if gene[i] == "1"
        ]
        gene_train = self.x_train.loc[:, self.x_train.columns.isin(features)]
        gene_test = self.x_test.loc[:, self.x_test.columns.isin(features)]

        neigh = KNeighborsClassifier()
        neigh.fit(gene_train, self.y_train.to_numpy().ravel())
        gene_pred = neigh.predict(gene_test)

        return accuracy_score(
            y_true=self.y_test.to_numpy().ravel(), y_pred=gene_pred, normalize=True
        )
```

Selection Algorithm

To select the best genes, we use the Tournament Selectin algorithm as Code Cell 3 shows. We utilised a `tournament_size` of 100.

```python
def tournament_selection(solutions_w_fitness: list[Solution], tournament_size: int):
    mating_pool = []
    for _ in solutions_w_fitness:
        best = Solution("None", 0)
        for _ in range(tournament_size):
            curr_element: list[Solution] = random.choice(solutions_w_fitness)
             if curr_element.fitness > best.fitness:
                best = curr_element
        mating_pool.append(best)
    return mating_pool
```

Crossover

Finally, once the selection is finished we perform Uniform Crossover over the selected parents. We utilised a `mutation_chance` of 0.05% and a `crossover_rate` of 85%. Each parent is equally likely to pass on a gene to a given offspring.

```python
def uniform_crossover(
    parent_a: Solution,
    parent_b: Solution,
    mutation_chance: float = 0.05,
    crossover_rate: float = 0.85,
):
    if random.uniform(0, 1) > crossover_rate:
        return parent_a, parent_b

    for i in range(NUM_FEATURES):
        if random.uniform(0, 1) <= mutation_chance:
            inverse = "0" if parent_a.gene[i] == "1" else "1"
            parent_a.gene = parent_a.gene[:i] + inverse + parent_a.gene[i + 1 :]
            parent_a.fitness = None
        if random.uniform(0, 1) <= mutation_chance:
            inverse = "0" if parent_a.gene[i] == "1" else "1"
            parent_b.gene = parent_b.gene[:i] + inverse + parent_b.gene[i + 1 :]
            parent_b.fitness = None

    mask = random.choices(["A", "B"], k=NUM_FEATURES)
    inv_mask = ["B" if k == "A" else "A" for k in mask]
    child1 = []
    child2 = []
    for i in range(NUM_FEATURES):
        if mask[i] == "A":
            child1.append(parent_a.gene[i])
        else:
            child1.append(parent_b.gene[i])
        if inv_mask[i] == "A":
            child2.append(parent_a.gene[i])
        else:
            child2.append(parent_b.gene[i])
    return Solution("".join(child1)), Solution("".join(child2))
```

Results

After running the Genetic Algorithm (GA) for 50 generations with a population size of 5000 we achieved the following results:

The best solution the GA found was utilising the following features: mean perimeter, mean smoothness, mean concavity, mean concave points, texture error, perimeter error, area error, compactness error, concave points error, symmetry error, fractal dimension error, worst perimeter, worst smoothness, worst concave points and worst fractal dimension.

As shown in table 1, the genetic algorithm provided a 2.6 % accuracy gain over the baseline. The execution time was significantly higher, however it is important to mention that the best result was achieved in an early generation, as such the number of generations was configured to be high so as to perform a more exhaustive search.

Feature Selection Method	# of Features	Accuracy Test Set (%)	Execution time (s)
Without Selection	30	92.1	3.8
Genetic Algorithm	15	94.7	1385.7