Artificial Intelligence - Homework 2

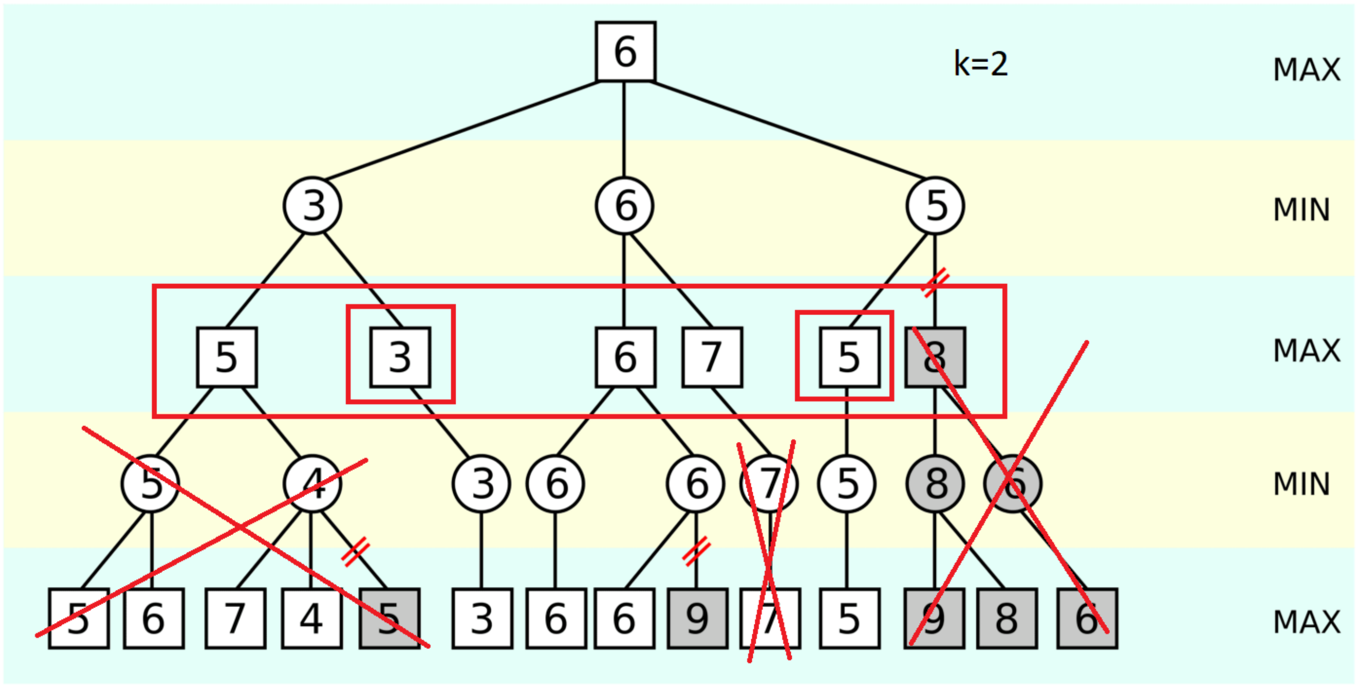
Domenico Santone - mat. 288640

Chess

Algorithmic Part

The selected algorithm from the 3 proposed is the **MiniMaxBestK** and is possible to parametrize the k value to select the best value.

The idea of the algorithm is to select the best k nodes at level l, which is parametrized; and then explore only the selected nodes at level L of the minimax implementing also alpha-beta pruning.



In the previous image, you can see that, for example, selecting the best k moves and using alpha-beta pruning can avoid the computation of a big part of the tree (the branches crossed).

Dataset

The dataset is obtained with a Stockfish evaluation at depth 22 associated with 64 features representing the board and is composed of over 1 million evaluated boards.

Each feature has a negative value if the color of the piece is black and a positive value otherwise, the value associated with the pieces are the following:

- 1 for the **pawns**
- 2 for the **knights**
- 3 for the **bishops**
- 4 for the **rooks**
- 5 for the **queen**
- 6 for the **king**

Regressor

The regressor chosen is **HistGradientBoostingRegressor** (Histogram-based Gradient Boosting Regression Tree), this estimator is much faster than `GradientBoostingRegressor` for big datasets (n_samples >= 10 000) and fits perfectly the use case.

Boosting refers to a class of ensemble learning algorithms that add tree models to an ensemble sequentially. Each tree model added to the ensemble attempts to correct the prediction errors made by the tree models already present in the ensemble.

Gradient boosting is a generalization of boosting algorithms like AdaBoost to a statistical framework that treats the training process as an additive model and allows arbitrary loss functions to be used, greatly improving the capability of the technique. As such, gradient boosting ensembles are the go-to technique for most structured (e.g. tabular data) predictive modeling tasks.

Although gradient boosting performs very well in practice, the models can be slow to train. This is because trees must be created and added sequentially, unlike other ensemble models like random forest where ensemble members can be trained in parallel, exploiting multiple CPU cores. As such, a lot of effort has been put into techniques that improve the efficiency of the gradient boosting training algorithm.

One aspect of the training algorithm that can be accelerated is the construction of each decision tree, the speed of which is bounded by the number of examples (rows) and number of features (columns) in the training dataset. Large datasets, e.g. tens of thousands of examples or more, can result in the very slow construction of trees as split points on each value, for each feature must be considered during the construction of the trees.

The construction of decision trees can be sped up significantly by reducing the number of values for continuous input features. This can be achieved by discretization or binning values into a fixed number of buckets. This can reduce the number of unique values for each feature from tens of thousands down to a few hundred. This allows the decision tree to operate upon the ordinal bucket (an integer) instead of specific values in the training dataset. This coarse approximation of the input data often has little impact on model skill, if not improves the model skill, and dramatically accelerates the construction of the decision tree.

Additionally, efficient data structures can be used to represent the binning of the input data; for example, histograms can be used and the tree construction algorithm can be further tailored for the efficient use of histograms in the construction of each tree.

As such, it is common to refer to a gradient boosting algorithm supporting "*histograms*" in modern machine learning libraries as a **histogram-based gradient boosting**.

Training phase

As you can see in the `training.py` file, after reading the file I proceeded to normalize the data using `StandardScaler`.

The transformation is given by

$z = (x - u)/s$

Where u is the mean of the training samples or zero if `with_mean=False`, and s is the standard deviation of the training samples or one if `with_std=False`.

After performing several tries the best result was achieved with the following hyperparameters:

- *early_stopping*=True,
- *l2_regularization*=0.3
- *max_iter*=300
- *max_leaf_nodes*=None

Report

In the following tables, you can see several tries against other algorithms and heuristics

OldCheck is the previous version of the minimax algorithm (the one from homework 1).

The BestK is the new algorithm and is followed by the name of the used Heuristic (i.e. BestKCheck means BestK algorithm with Check Heuristic).

The new Heuristic is in the file `PredictHeuristic` and it uses the regressor to predict the value of the board, I also implemented a simple mechanism to avoid reloading the model each time but storing it inside the class to improve performance.

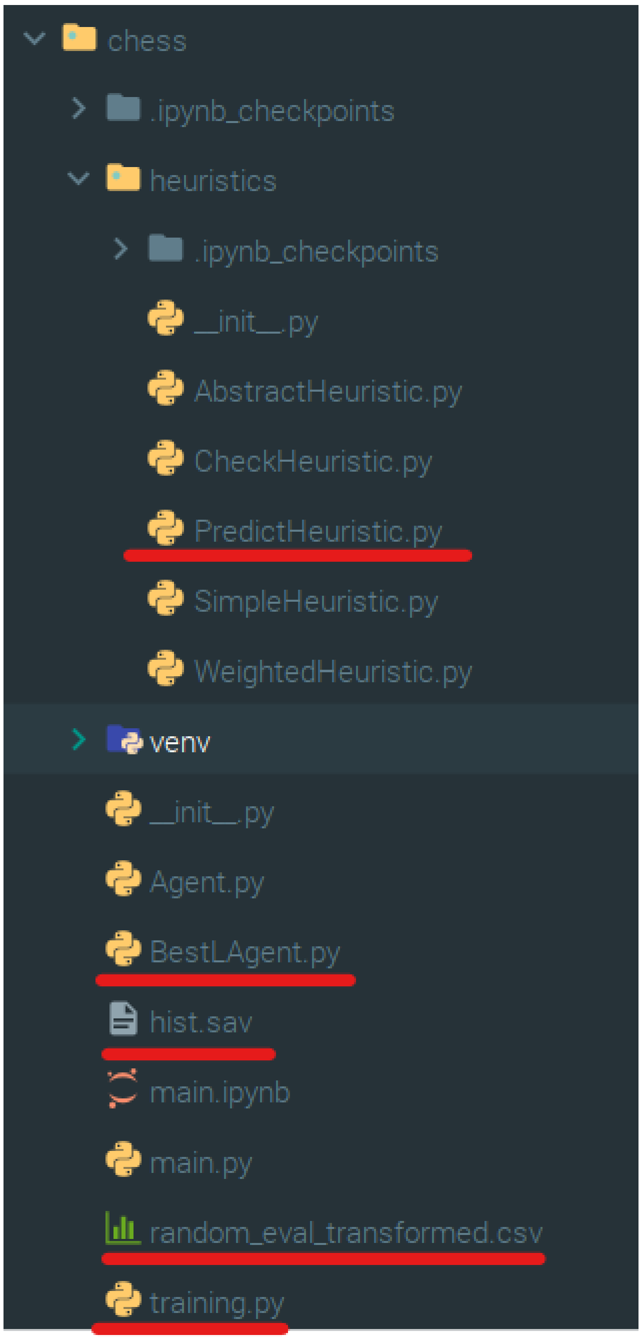
White | Black | Draw	OldCheck (depth=1)
BestKPredict (k=1, depth=1)	10% |8% | 82%
BestKPredict (k=2, depth=1)	13% | 11% | 76%
BestKPredict (k=3, depth=1)	10% | 11% | 79%
BestKPredict (k=1, depth=3)	0% | 21% | 79%
BestKPredict (k=2, depth=3)	0% | 20% | 80%
BestKPredict (k=3, depth=3)	0% | 23% | 77%

White | Black | Draw	BestKCheck (k=1, depth=3)	BestKCheck (k=1, depth=6)
BestKPredict (k=1, depth=1)	22% | 0% | 78%	12% | 0% | 88%
BestKPredict (k=2, depth=1)	24% | 0% |76%	10% | 0% | 90%
BestKPredict (k=3, depth=1)	28% | 0% | 72%	9% | 0% | 91%
BestKPredict (k=1, depth=3)	0% | 0% | 100%	0% | 0% | 100%
BestKPredict (k=2, depth=3)	0% | 0% | 100%	0% | 0% | 100%
BestKPredict (k=3, depth=3)	0% | 0% | 100%	0% | 0% | 100%

White | Black | Draw	BestKWeighted (k=1, depth=3)	BestKWeighted (k=1, depth=6)
BestKPredict (k=1, depth=1)	0% | 0% | 100%	6% | 0% | 94%
BestKPredict (k=2, depth=1)	1% | 0% |99%	18% | 0% | 82%
BestKPredict (k=3, depth=1)	0% | 0% | 100%	19% | 0% | 81%
BestKPredict (k=1, depth=3)	0% | 0% | 100%	0% | 0% | 100%
BestKPredict (k=2, depth=3)	0% | 0% | 100%	0% | 0% | 100%
BestKPredict (k=3, depth=3)	0% | 0% | 100%	0% | 0% | 100%

White | Black | Draw	BestKSimple (k=1, depth=3)	BestKSimple (k=1, depth=6)
BestKPredict (k=1, depth=1)	15% | 0% | 85%	8% | 0% | 92%
BestKPredict (k=2, depth=1)	17% | 0% | 83%	14% | 0% | 86%
BestKPredict (k=3, depth=1)	19% | 0% | 81%	13% | 0% | 87%
BestKPredict (k=1, depth=3)	0% | 0% | 100%	0% | 0% | 100%
BestKPredict (k=2, depth=3)	0% | 0% | 100%	0% | 0% | 100%
BestKPredict (k=3, depth=3)	0% | 0% | 100%	0% | 0% | 100%

Code Structure



The new files are underlined:

- `PredictHeuristic.py` -> the class that implements the loading of the model and returns the result to the minimax algorithm. In order to make the prediction possible it also contains some helper functions to translate a board into a flat array of 64 elements.
- `BestLAgent.py` -> the class that implements the new version of the algorithm MiniMaxBest

- `hist.sav` -> is the file that contains the model used for the prediction part (is a heavy file so the first evaluation and only the first as mentioned before, of the agent can take a little bit of time to load the model).
- `random_eval_transformed.csv`-> is the dataset used for the training of the algorithm
- `training.py` -> is the script used to train the model (please avoid running it if you don't want to retrain the whole model and overwrite the one in the directory)