Exercise 2

Random Forest

Lukas Sichert -12114770

Kristof Dadic - 12105475

Florian Engl - 12102619

Regression Tree Code Overview

```
class RegressionTree:

    class Node:
        def __init__(self, col=-1, threshold=None, result = None, left=None, right=None):…

    def __init__(self, max_depth=-1, min_samples_split=2, max_features=None, random_state=None):…

    def build_tree(self, X, y, score, depth):…

    def mse(self, y, y_left, y_right):…

    def std(self, y, y_left, y_right):…

    def fit(self, X, y, score=None):        …

    def regress(self, observation, node):…

    def predict(self, features):…
```

Regression Tree - build_tree function

```
def build_tree(self, X, y, score, depth):
    if len(y) < self.min_samples_split:
        # print("Not enough samples to split")
        return self.Node(result=np.mean(y))
    if depth == 0:
        # print("Max depth reached")
        return self.Node(result=np.mean(y))
    best_feature, best_threshold, best_score = None, None, float('inf')
    dim=X.shape[1]
    if self.max_features==None:
        sel_features=range(dim)
    elif self.max_features=="sqrt":
        sel_features=self.rng.choice(range(dim), int(np.sqrt(dim)))
    elif self.max_features=="log2":
        sel_features=self.rng.choice(range(dim),int(np.log2(dim)))
    elif isinstance(self.max_features, int):
        sel_features=self.rng.choice(range(dim),self.max_features)
    for feature in sel_features:
        feature_values = X[:, feature]
        for threshold in np.unique(feature_values):
            y_left = y[feature_values < threshold]
            y_right = y[feature_values >= threshold]

            if len(y_left) == 0 or len(y_right) == 0:
                continue
            current_score = score(y, y_left, y_right)
            if current_score < best_score:
                best_score = current_score
                best_threshold = threshold
                best_feature = feature
    if best_threshold is None:
        # print("No best threshold found")
        return self.Node(result=np.mean(y))
    else:
        left = self.build_tree(X[X[:, best_feature] < best_threshold], y[X[:, best_feature] < best_threshold], score, depth - 1)
        right = self.build_tree(X[X[:, best_feature] >= best_threshold], y[X[:, best_feature] >= best_threshold], score, depth - 1)
        return self.Node(col=best_feature, threshold=best_threshold, left=left, right=right)
```

Random Forest Code Overview

```
class ourRandomForestRegressor():

    def __init__(self, nb_trees=40,  nb_samples = "Full", max_depth=-1, max_workers=-1, random_state=None, boot_type = True, min_samples_split=2, max_features=None):…

    def fit(self, X, y):…

    def train_tree(self, data, random_state):…

    def predict(self, feature):…

    def get_params(self, deep=True):…

    def set_params(self, **params):…
```

Random Forest - fit function

```
def fit(self, X, y):
    if type(X) == np.ndarray:…
    else:…
    length = len(data)
    if isinstance(self.nb_samples, float) and 0<self.nb_samples<1:…
    elif isinstance(self.nb_samples, int):…
    else: nb_samples = length

    with ProcessPoolExecutor(max_workers=self.max_workers) as executor:
        indices = np.arange(length)
        #chooses random indices to bootstrap the data
        rand_ind = [self.rng.choice(indices, size=nb_samples, replace=self.boot_type) for _ in range(self.nb_trees)]
        #extracts the data for the chosen indices
        bootstrap_data = [[data[i] for i in ind] for ind in rand_ind]
        #initializes randomly generated states for each tree
        random_states = self.rng.integers(low=0, high=1e6, size=self.nb_trees)
        #builds the trees
        self.trees = list(executor.map(self.train_tree, bootstrap_data, random_states))
```

LLM Tree - _build_tree function

```python
def _build_tree(self, X, y, depth=0):
    if len(y) < self.min_samples_split or (self.max_depth is not None and depth >= self.max_depth):
        return np.mean(y)

    best_split = self._find_best_split(X, y)
    if best_split is None:
        return np.mean(y)

    left_idx = best_split['left_idx']
    right_idx = best_split['right_idx']
    left_subtree = self._build_tree(X[left_idx], y[left_idx], depth + 1)
    right_subtree = self._build_tree(X[right_idx], y[right_idx], depth + 1)
    return {
        'feature_idx': best_split['feature_idx'],
        'threshold': best_split['threshold'],
        'left': left_subtree,
        'right': right_subtree,
    }
```

```python
def _find_best_split(self, X, y):
    best_split = None
    best_cost = float('inf')

    for feature_idx in range(X.shape[1]):
        thresholds = np.unique(X[:, feature_idx])
        for threshold in thresholds:
            left_idx = np.where(X[:, feature_idx] <= threshold)[0]
            right_idx = np.where(X[:, feature_idx] > threshold)[0]

            if len(left_idx) == 0 or len(right_idx) == 0:
                continue

            cost = self._compute_split_cost(y[left_idx], y[right_idx])
            if cost < best_cost:
                best_cost = cost
                best_split = {
                    'feature_idx': feature_idx,
                    'threshold': threshold,
                    'left_idx': left_idx,
                    'right_idx': right_idx,
                }
```

LLM - Random_Forest fit function

```python
def fit(self, X, y):
    # Convert to NumPy arrays
    X = np.array(X)
    y = np.array(y)

    n_samples, n_features = X.shape

    # Interpret `max_features`
    if isinstance(self.max_features, str):
        if self.max_features == "sqrt":
            self.max_features = int(np.sqrt(n_features))
        elif self.max_features == "log2":
            self.max_features = int(np.log2(n_features))
        else:
            raise ValueError(f"Invalid value for max_features: {self.max_features}")
    elif self.max_features is None:
        self.max_features = n_features
    elif isinstance(self.max_features, (int, float)):
        self.max_features = int(self.max_features)

    for _ in range(self.n_estimators):
        # Sample data with replacement (Bootstrap)
        indices = self.rng.choice(n_samples, n_samples, replace=True)
        X_sample = X[indices]
        y_sample = y[indices]

        # Randomly select feature subset
        feature_indices = self.rng.choice(n_features, self.max_features, replace=False)
        self.feature_subsets.append(feature_indices)

        # Train a decision tree on the bootstrap sample
        tree = DecisionTreeRegressor(max_depth=self.max_depth, min_samples_split=self.min_samples_split)
        tree.fit(X_sample[:, feature_indices], y_sample)
        self.trees.append(tree)
```

Key Differences

- random feature selection:
  - we select a random feature subset at each tree node
  - LLM selects the random subset once for the whole tree
- LLM did not parallelize, leading to worse performance
- bootstrapping samples:
  - we always bootstrap and control replacement via boot_type = TRUE/FALSE
  - LLM does not bootstrap at all
  - scikit-learn bootstraps either with replacement or not at all, controlled via boot_type

Tuning Results - Concrete Dataset

```
Our RF
Scoring: RSE
Best score: -0.0926801555831973
best params: OrderedDict([('model__boot_type', False), ('model__max_depth', 40), ('model__max_features', 'log2'),
                          ('model__min_samples_split', 2), ('model__nb_samples', 'Full'), ('model__nb_trees', 40)])


LLM RF
Scoring: RSE
Best score: -0.09835495652703449
best params: OrderedDict({'model__max_depth': None, 'model__max_features': None, 'model__min_samples_split': 2, 'model__n_estimators': 100})


Scikit-Learn RF
Scoring: RSE
Best score: -0.08167004287652026
best params: OrderedDict([('model__bootstrap', False), ('model__max_depth', 30), ('model__max_features', 'log2'),
                          ('model__min_samples_split', 2), ('model__n_estimators', 300)])


kNN
Scoring: RSE
Best score: -0.19120078574345334
best params: OrderedDict({'model__n_neighbors': 5, 'model__p': 5, 'model__weights': 'distance'})
```

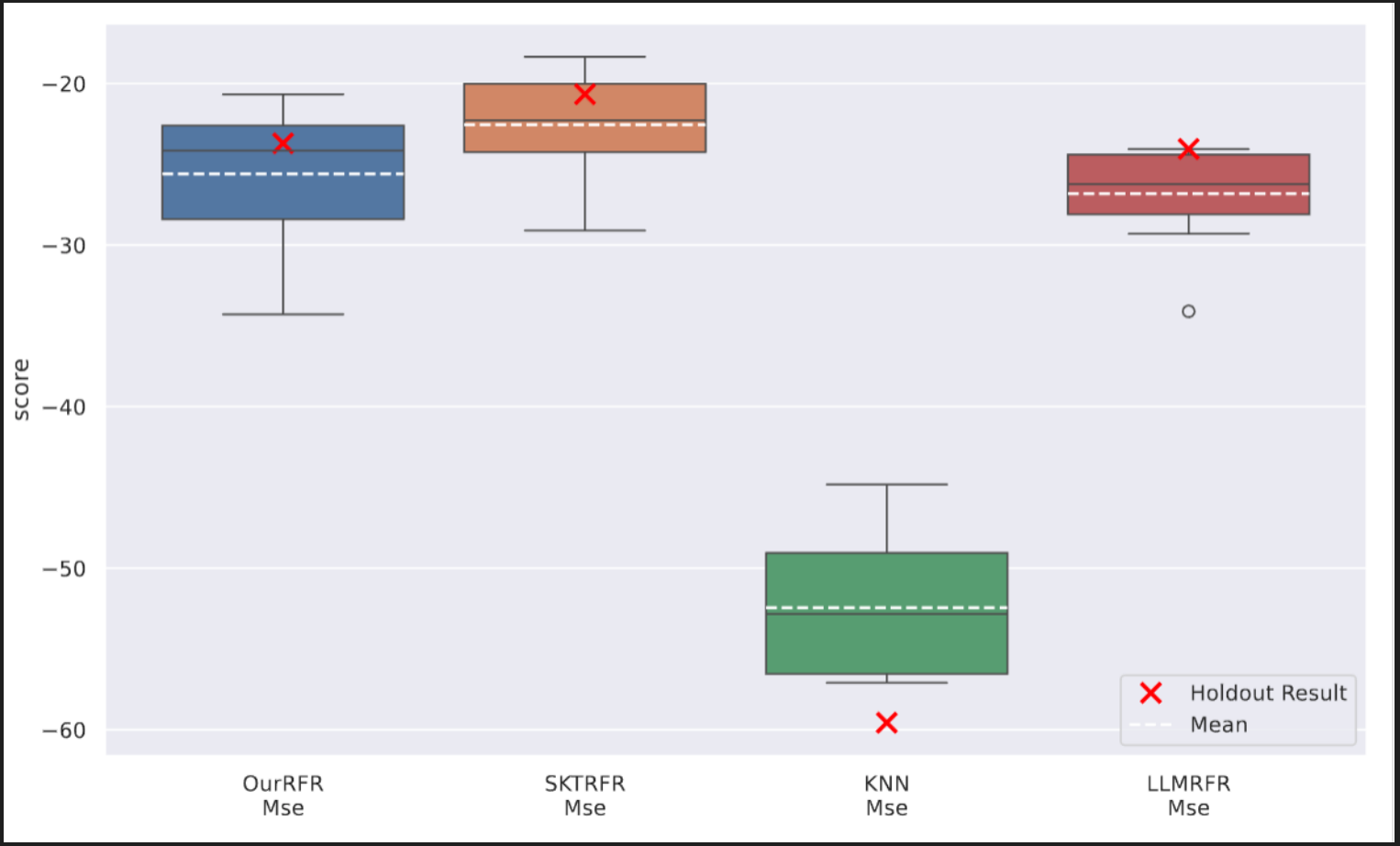
Tuning Results - Superconductivity Dataset

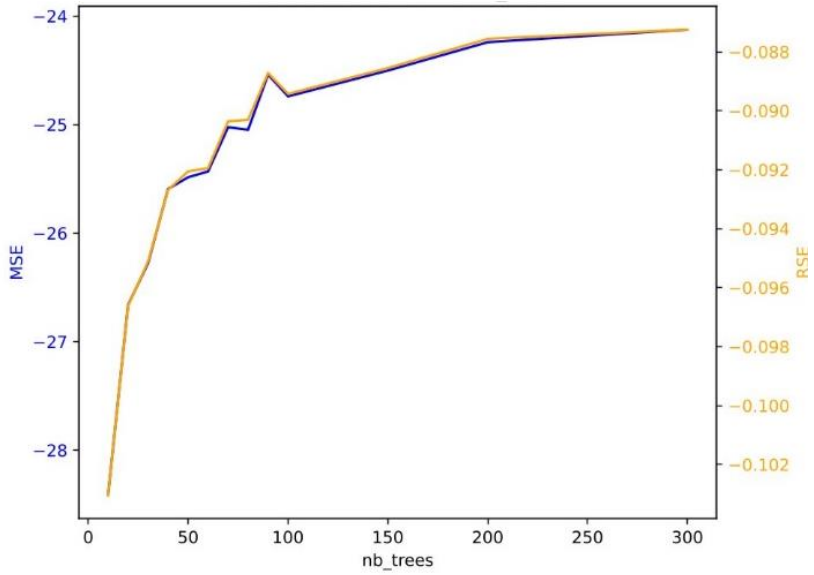
```
Our RF
Scoring: RSE
Best score: -0.07460708168885685
best params: OrderedDict({'model__boot_type': False, 'model__max_depth': 100, 'model__max_features': 'sqrt',
                          'model__min_samples_split': 2, 'model__nb_samples': 0.7, 'model__nb_trees': 150})


LLM RF
Scoring: RSE
Best score: -0.07880142280763472
best params: OrderedDict({'model__max_depth': 90, 'model__max_features': 'sqrt', 'model__min_samples_split': 4, 'model__n_estimators': 200})


Scikit-Learn RF
Scoring: RSE
Best score: -0.07425225089052202
best params: OrderedDict({'model__bootstrap': False, 'model__max_depth': 50,'model__max_features': 'sqrt',
                          'model__min_samples_split': 11, 'model__n_estimators': 80})


kNN
Scoring: RSE
Best score: -0.0869198050175312
best params: OrderedDict({'model__n_neighbors': 4, 'model__p': 1, 'model__weights': 'distance'})
```

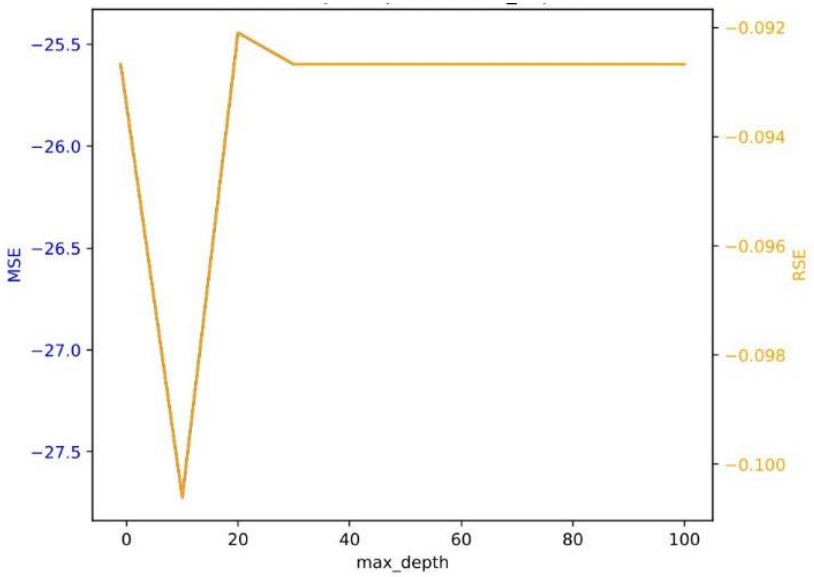
Cross Validation Results - Concrete Dataset

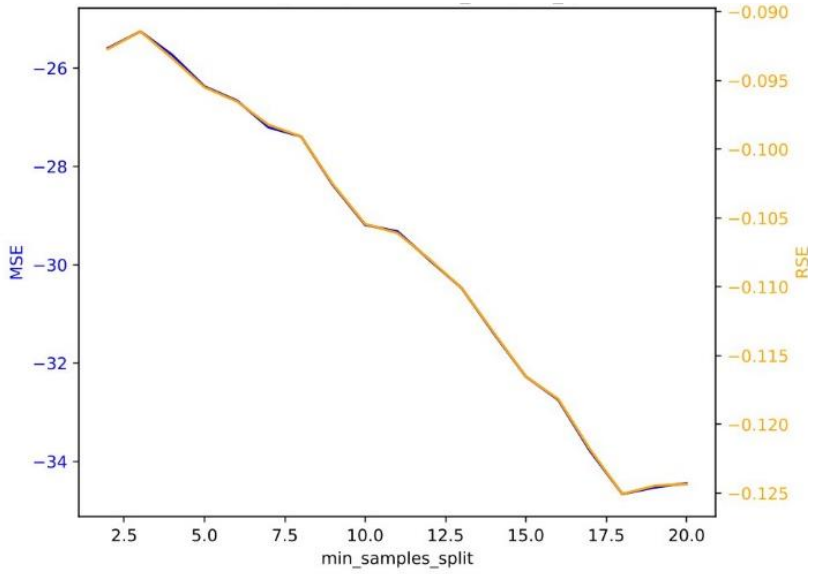
```
Relative Squared Error:
KNN Rse        -0.191201
LLMRFR Rse     -0.097498
OurRFR Rse     -0.092680
SKTRFR Rse     -0.081670

Mean Squared Error:
KNN Mse       -52.444999
LLMRFR Mse    -26.818862
OurRFR Mse    -25.597001
SKTRFR Mse    -22.552244
```

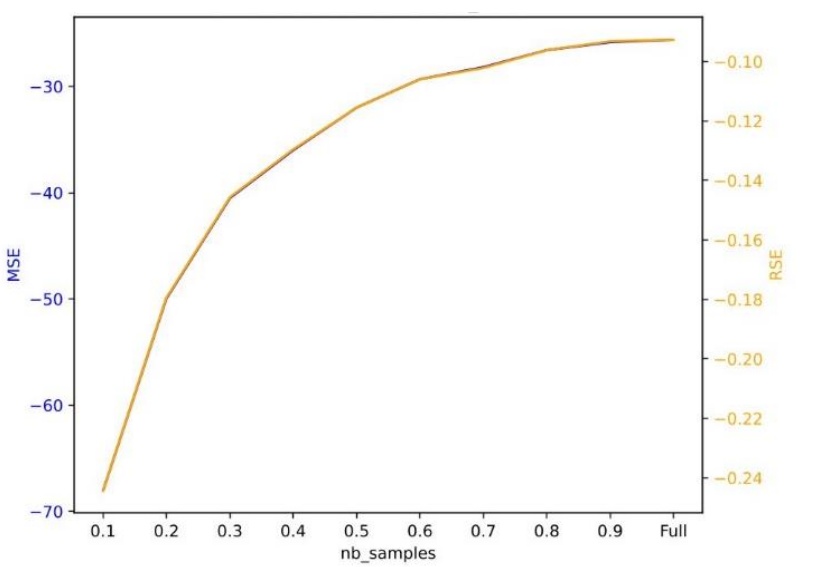
MSE Comparison - Concrete Dataset







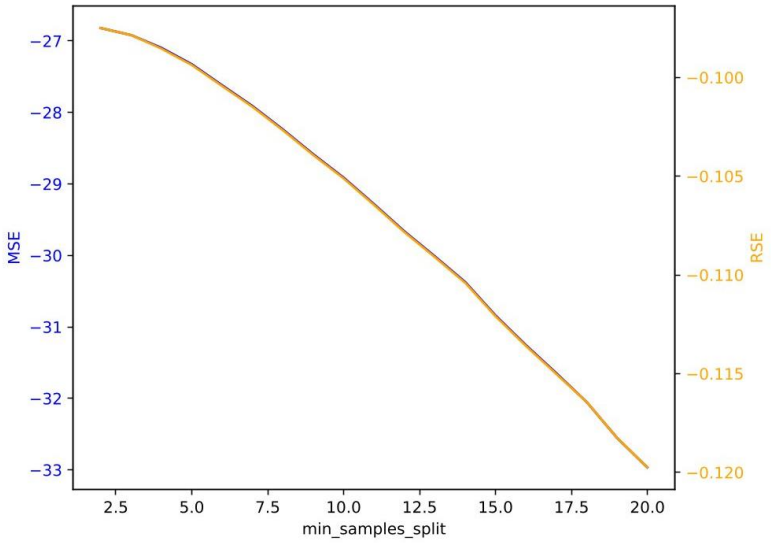


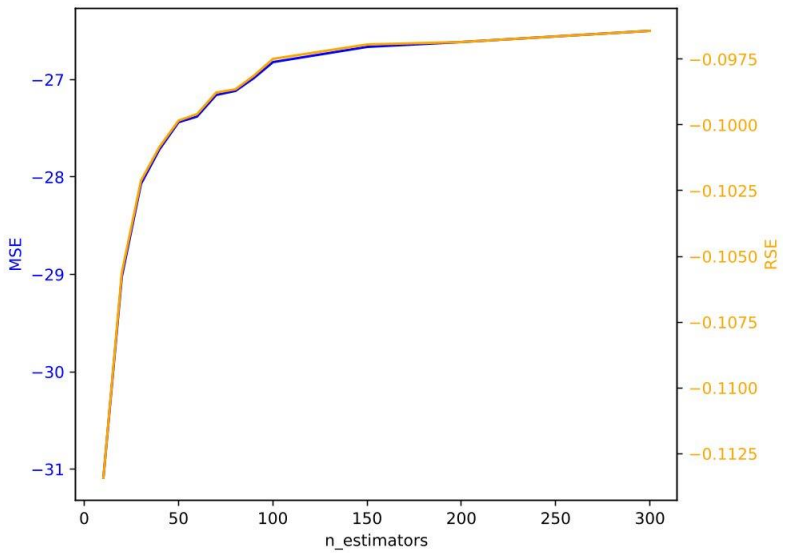


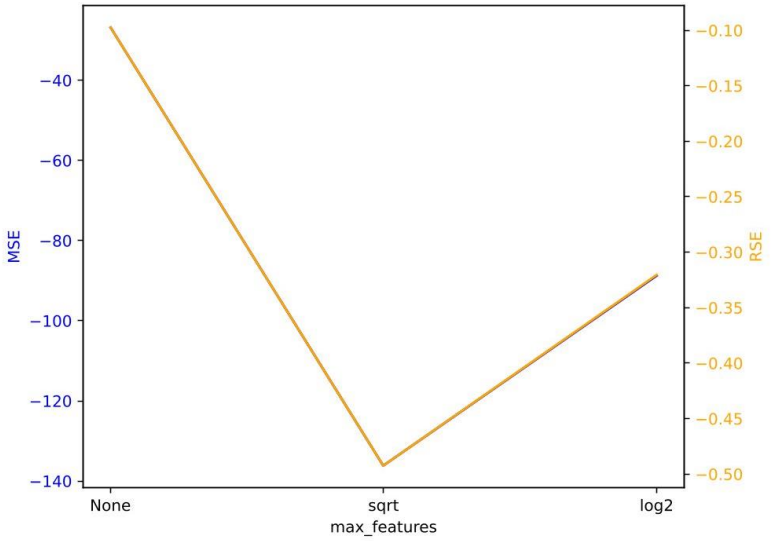
Sensitivity Plots

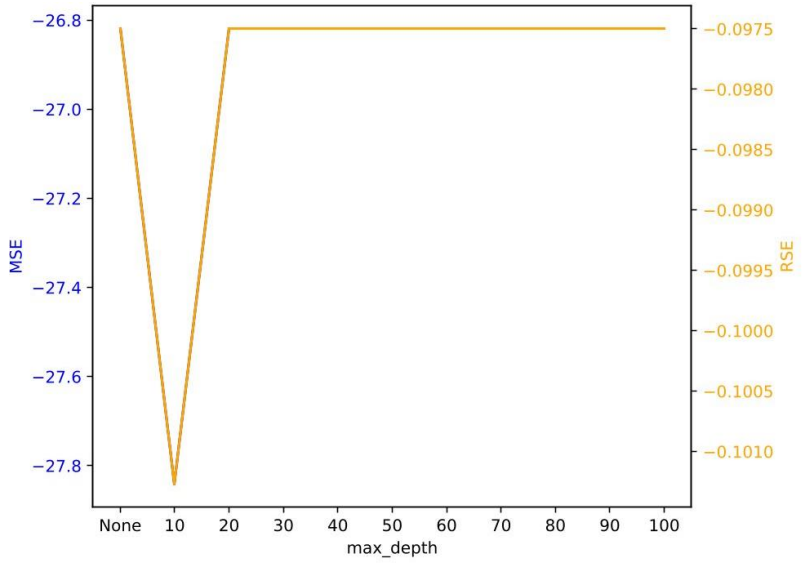
Concrete Dataset

Our Implementation





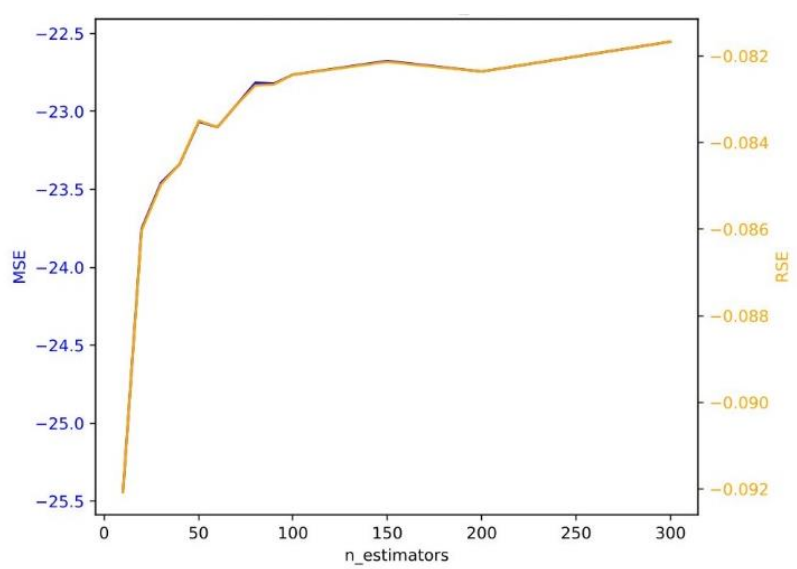


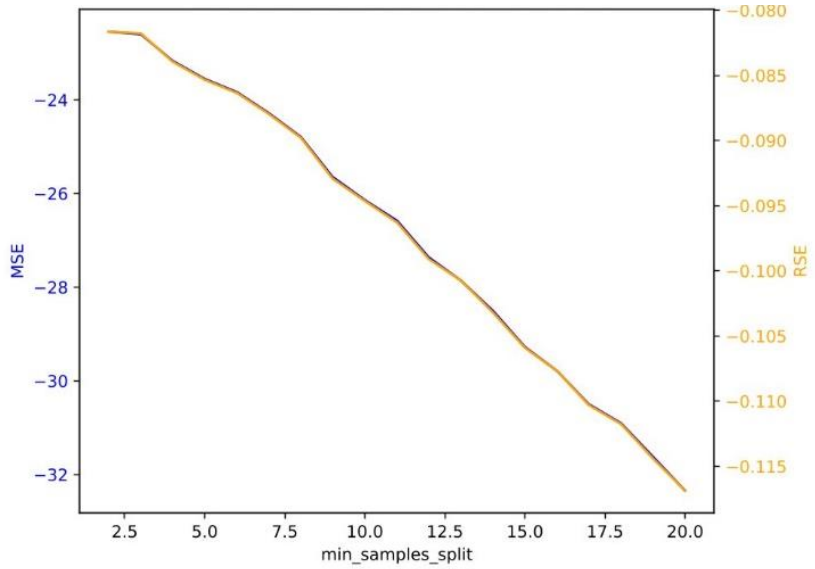


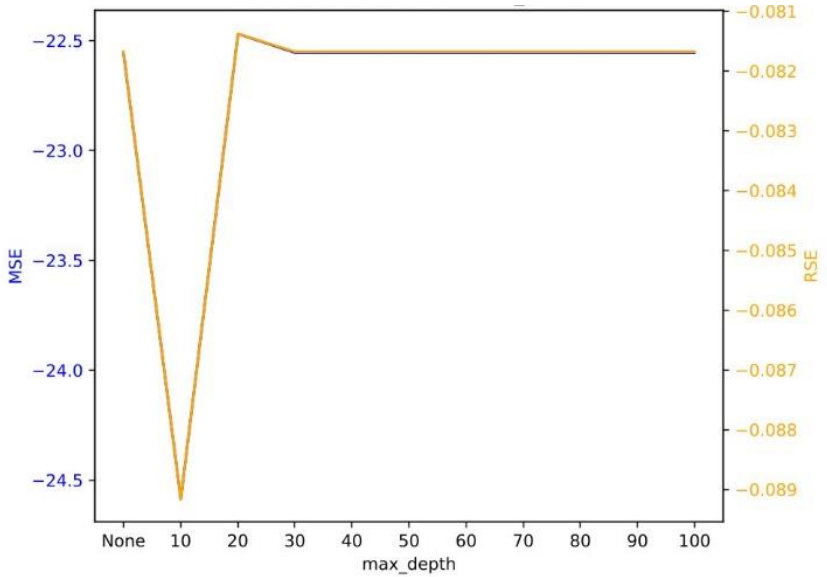
Sensitivity Plots

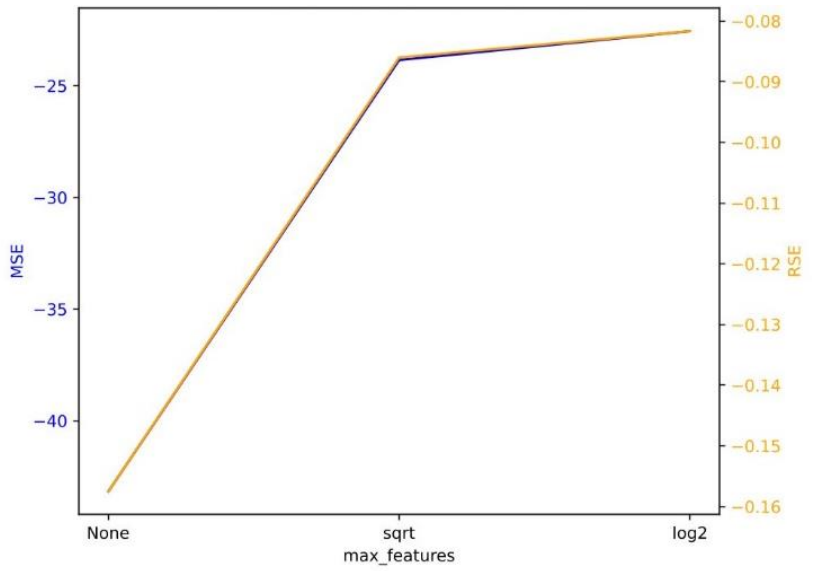
Concrete Dataset

LLM

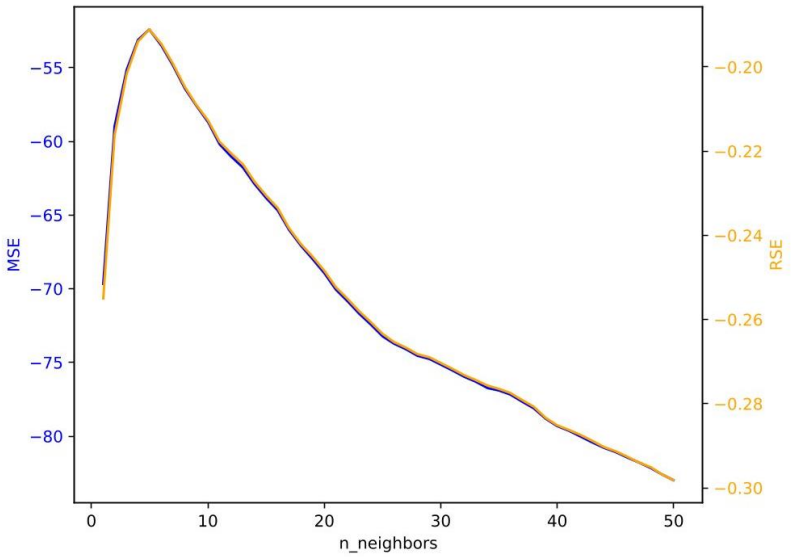


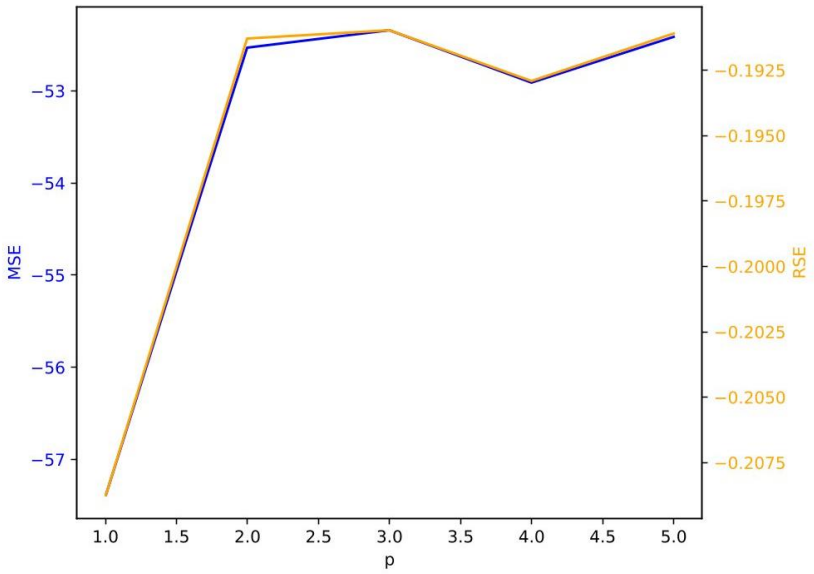


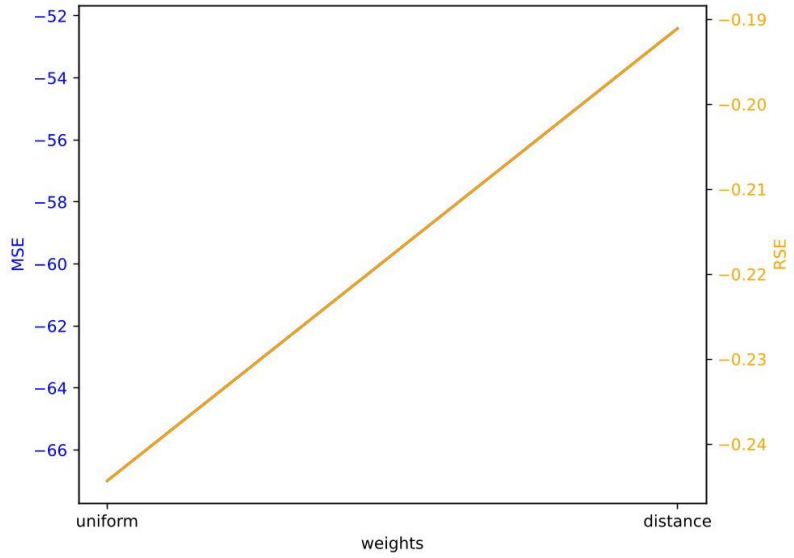




Sensitivity Plots
Concrete Dataset
scikit-rf







Sensitivity Plots
Concrete Dataset
kNN

Runtime - Concrete Dataset

model	train_time	pred_time
OurRFR	1.504795	0.023677
SKTRFR	0.414534	0.013145
KNN	0.002308	0.007700
LLMRFR	33.386977	0.035730

Cross Validation Results - Superconductivity Dataset

```
Relative Squared Error:
KNN Rse         -0.086920
OurRFR Rse      -0.075015
SKTRFR Rse      -0.074365

Mean Squared Error:
KNN Mse       -101.985703
OurRFR Mse     -88.010490
SKTRFR Mse     -87.240354
```

Cross Validation Results - Superconductivity

Runtime of models:

	model	train_time	pred_time
0	OurRFR	2340.603187	5.993693
1	SKTRFR	10.307848	0.134639
2	KNN	0.033743	2.491119
3	LLMRFR	8497.948970	6.141224

(no repetitions and few folds)

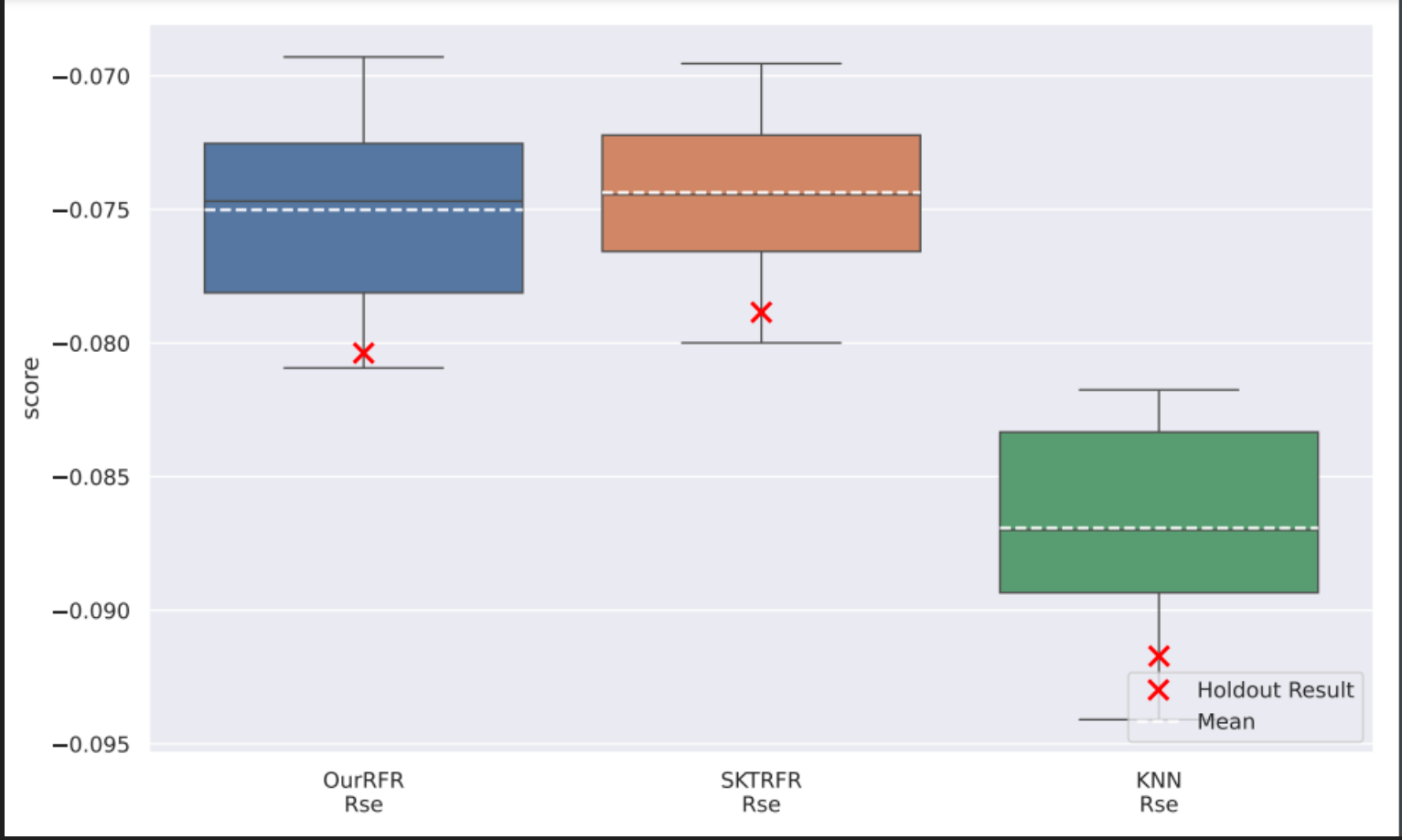
Mean Squared Error:

KNN Mse	-104.568489
LLMRFR Mse	-96.532095
OurRFR Mse	-91.647973
SKTRFR Mse	-91.450742

Relative Squared Error:

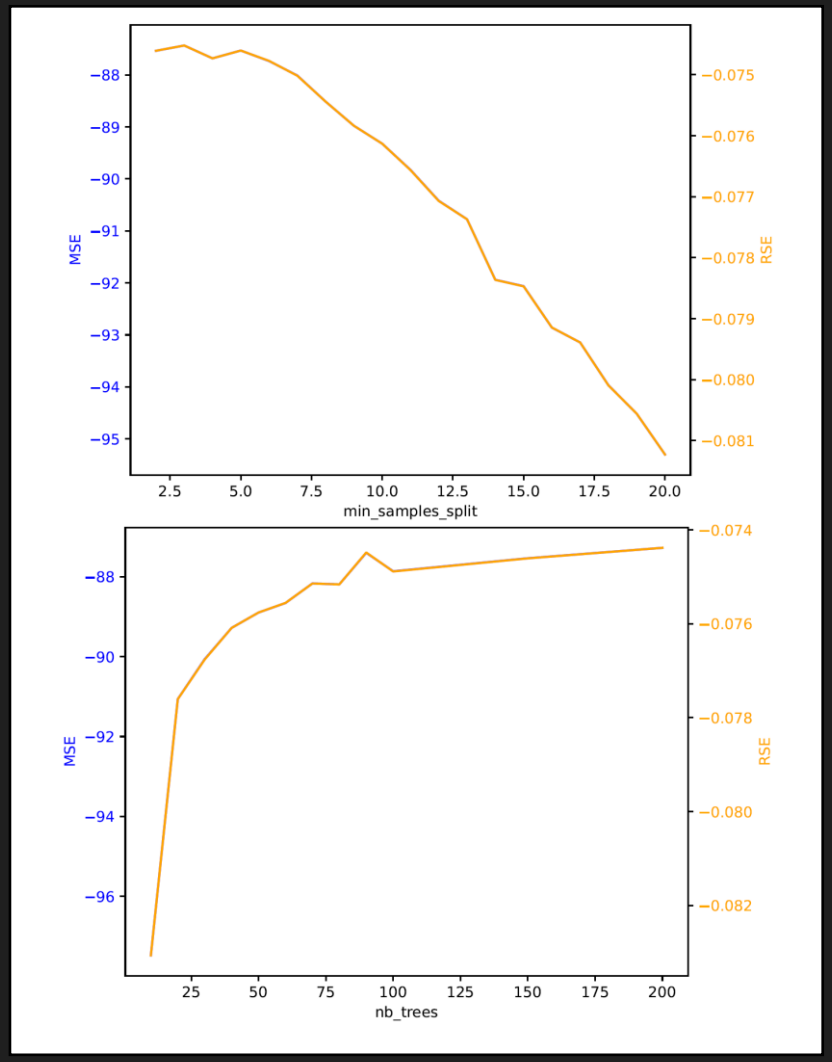
KNN Rse	-0.089120
LLMRFR Rse	-0.082268
OurRFR Rse	-0.078102
SKTRFR Rse	-0.077965

Relative Squared Error - Superconductivity Dataset

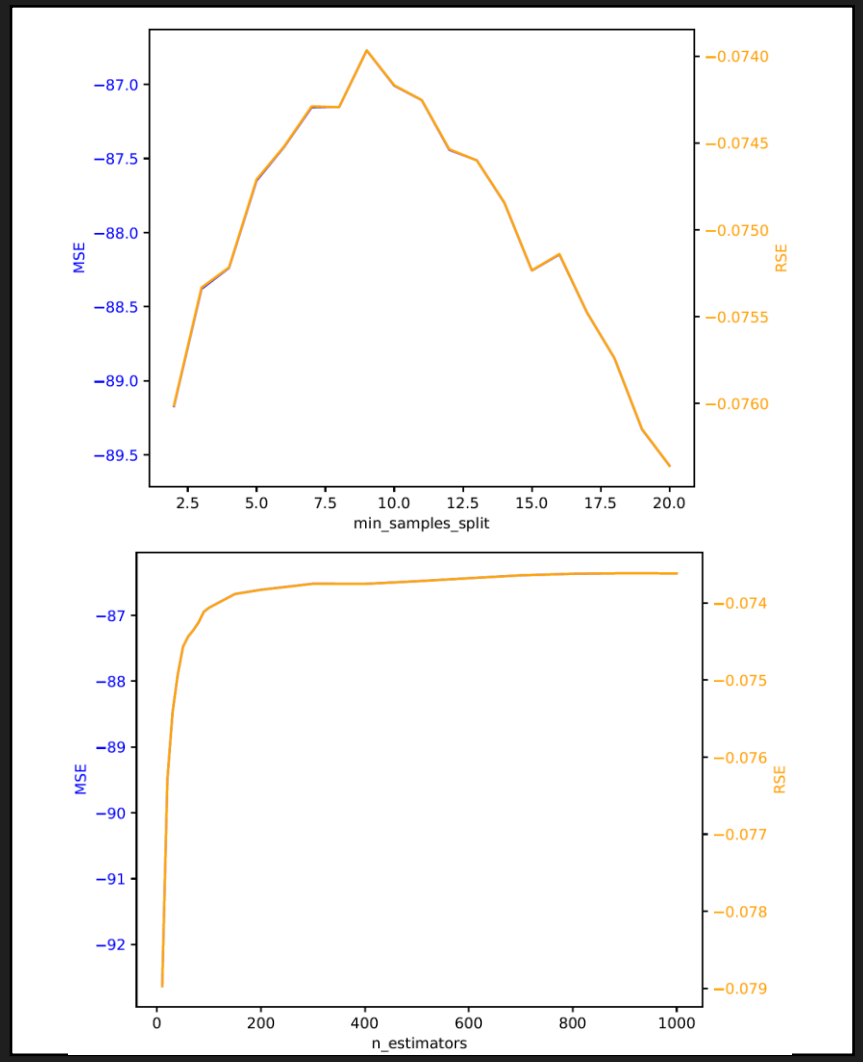


Sensitivity Plots – Superconductivity Dataset
min_samples_split, nb_estimators/trees

Our Implementation



Scikit RF



Runtime – Superconductivity Dataset

	model	train_time	pred_time
0	OurRFR	756.071104	4.227214
1	SKTRFR	7.741411	0.095570
2	KNN	0.011842	1.109954
3	LLMRFR	DNF	DNF

Conclusion – Key Takeaways

- Performance:
  - implementing the max_features functionality boosts both efficiency and effectiveness
    - especially choosing a random subset at every node
    - biggest difference between LLM and our/Scikit's RF
  - randomising the subsets of instances for each tree improves performance
  - min_samples_split can have a big influence depending on the implementation

- Efficiency:
  - parallelisation is necessary for application on bigger datasets
  - Scikit-Learn is very efficient; bigger difference in efficiency, not effectiveness
  - kNN is very fast, but lacks in effectiveness
    - even a single regression tree outperforms it
    - Interestingly the metric had a very big influence