Machine Learning project:
A classification task on heart conditions

Student: Samuele Ceol

Acknowledgments (Dataset creators):

1. Hungarian Institute of Cardiology. Budapest: Andras Janosi, M.D.
2. University Hospital, Zurich, Switzerland: William Steinbrunn, M.D.
3. University Hospital, Basel, Switzerland: Matthias Pfisterer, M.D.
4. V.A. Medical Center, Long Beach and Cleveland Clinic Foundation: Robert Detrano, M.D., Ph.D.
5. David W. Aha (aha '@' ics.uci.edu) (714) 856-8779

Rationale & Objective

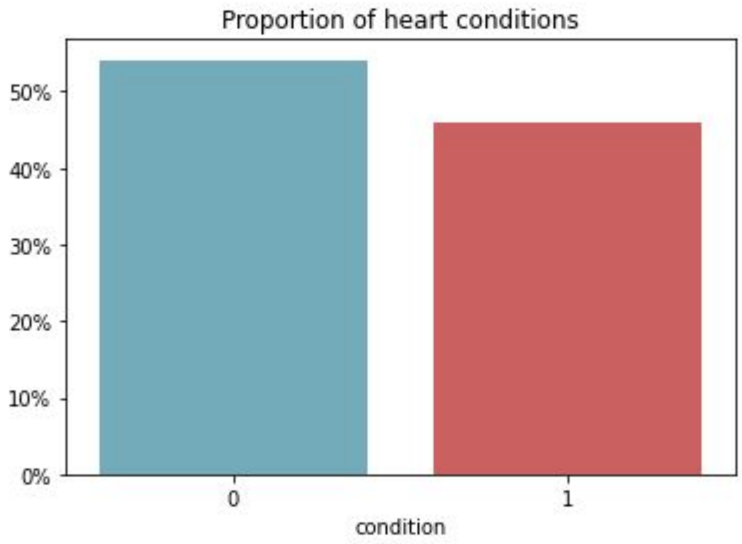
- Forming a deeper understanding of tree-based algorithms
- Developing a custom implementation of a Decision Tree (CART) and Random Forest algorithm
- Establishing a baseline and comparing it with the custom implementation
- Applying the algorithms to a relevant domain

The Dataset

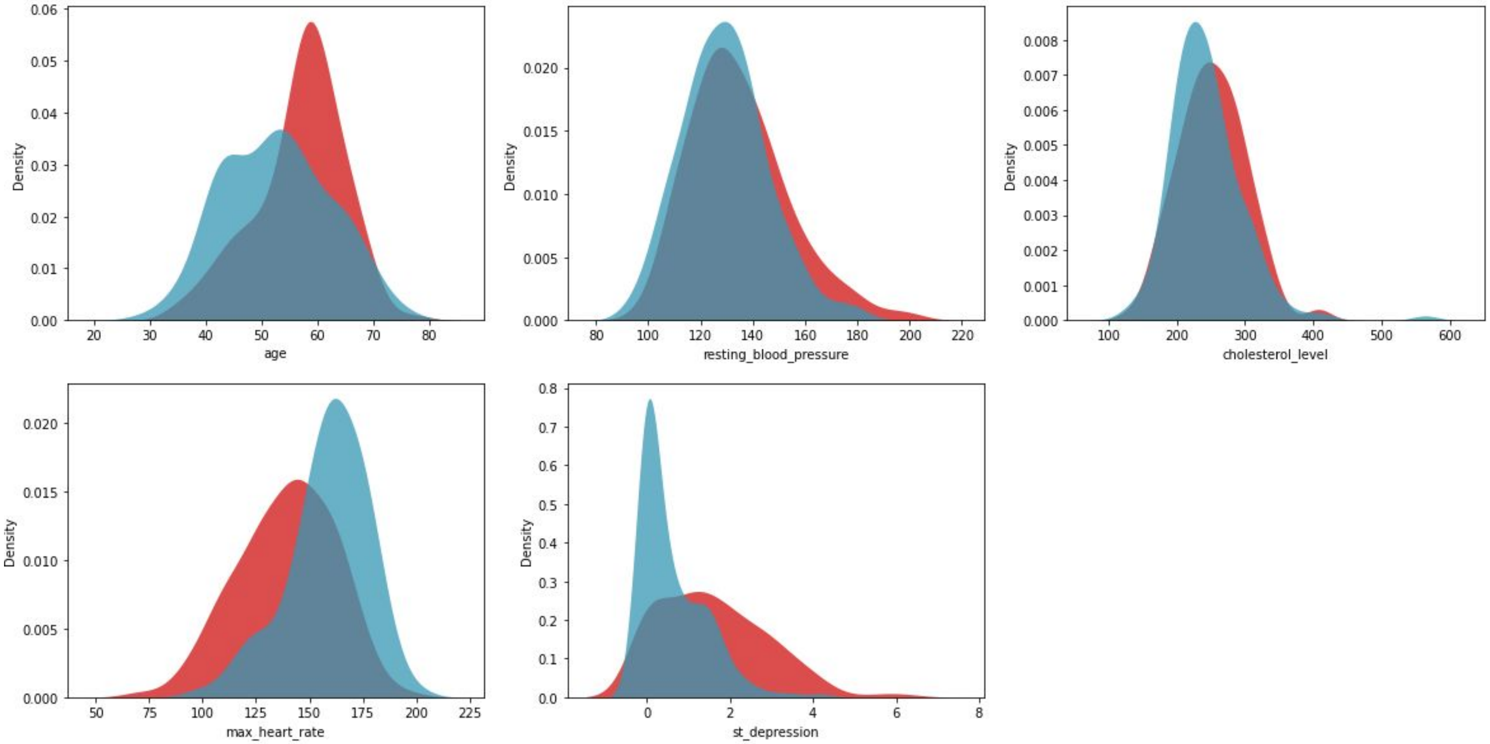
Dataset information

- Medical domain
- Cleveland Heart Disease dataset
- UCI Machine Learning Repository

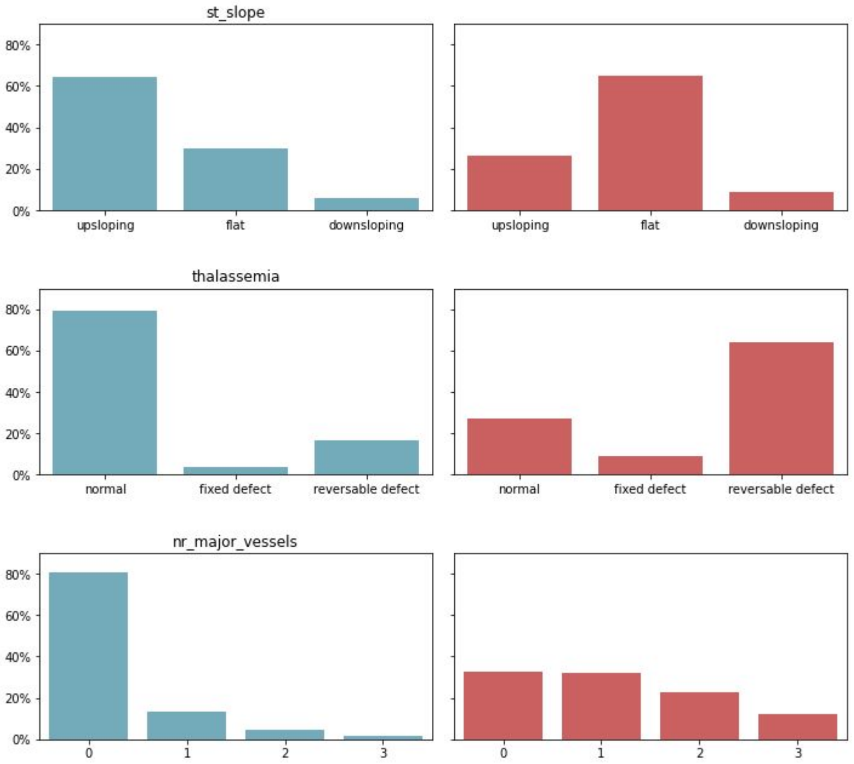
age	41
sex	2
chest_pain	4
resting_blood_pressure	50
cholesterol_level	152
fasting_blood_sugar	2
resting_ecg	3
max_heart_rate	91
exercise_induced_angina	2
st_depression	40
st_slope	3
nr_major_vessels	4
thalassemia	3
condition	2

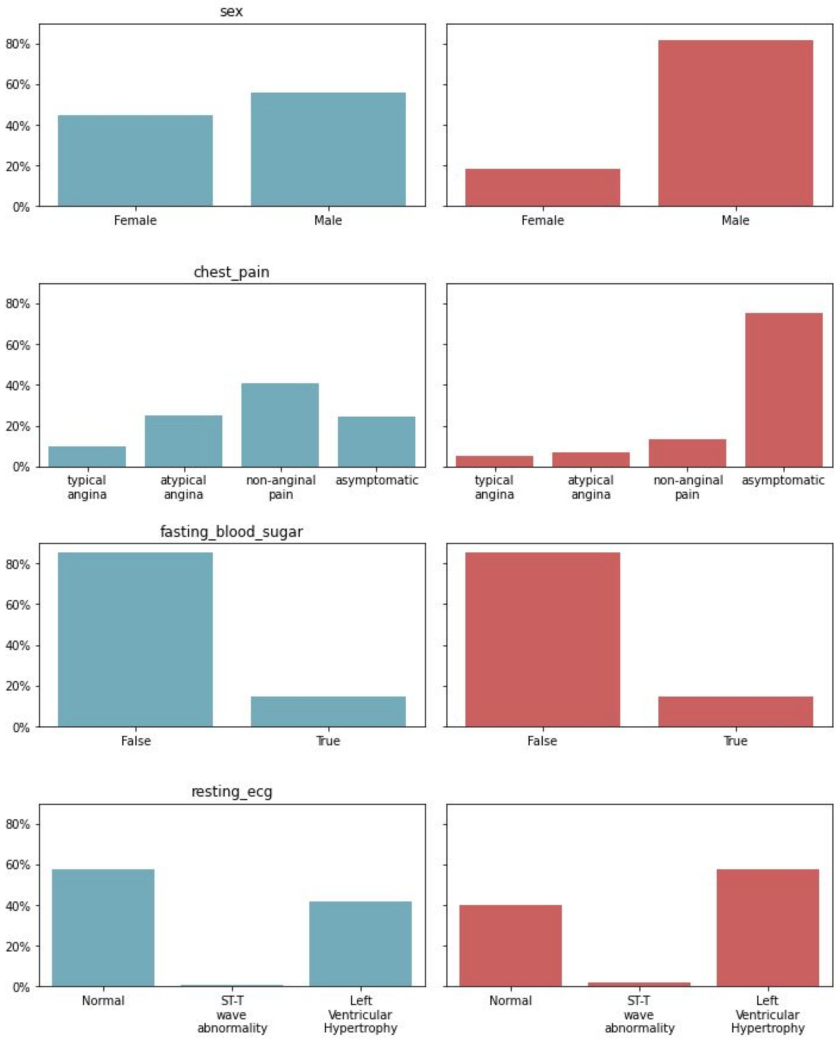


Numerical data



Categorical data





Data curation

Handling non-binary categorical data

- Categorical data can be problematic
- Encoding imposes an ordering that might not have been initially present
- Solution:
  - One-hot encoding

```
high_cardinality_cols = ['chest_pain', 'resting_ecg', 'st_slope', 'nr_major_vessels', 'thalassemia']
df = pd.get_dummies(df, columns=high_cardinality_cols, drop_first=True)
df
```

Data Curation - Handling missing values

- To build our models we cannot have missing values
- Solution:
    - Multiple imputation (by Chained Equations)

```
kernel = mf.KernelDataSet(df, save_all_iterations=True, random_state=42)
kernel.mice(5) # Nr of iterations
df = kernel.complete_data()

df.nr_major_vessels = df.nr_major_vessels.astype(int)
df.thalassemia = df.thalassemia.astype(int)
```

```
age                        0
sex                        0
chest_pain                 0
resting_blood_pressure     0
cholesterol_level          0
fasting_blood_sugar        0
resting_ecg                0
max_heart_rate             0
exercise_induced_angina    0
st_depression              0
st_slope                   0
nr_major_vessels           4
thalassemia                2
condition                  0
dtype: int64
```

Feature Scaling

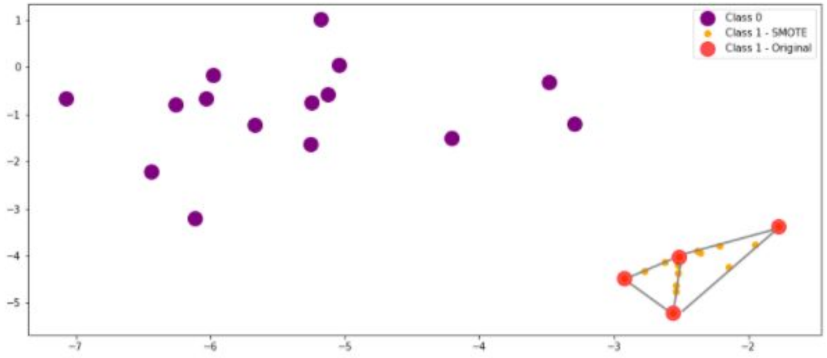
- Differing degrees of magnitude between features may affect the performance of some algorithms
- Solution:
  - Feature scaling
- Tree-based algorithms are unaffected by this problem, but feature scaling does not hurt them

$$X' = \frac{X - \mu}{\sigma} \qquad X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

```
cols_to_scale = ['age', 'resting_blood_pressure', 'cholesterol_level', 'max_heart_rate', 'st_depression']
ss = StandardScaler()
df[cols_to_scale] = ss.fit_transform(df[cols_to_scale])
```

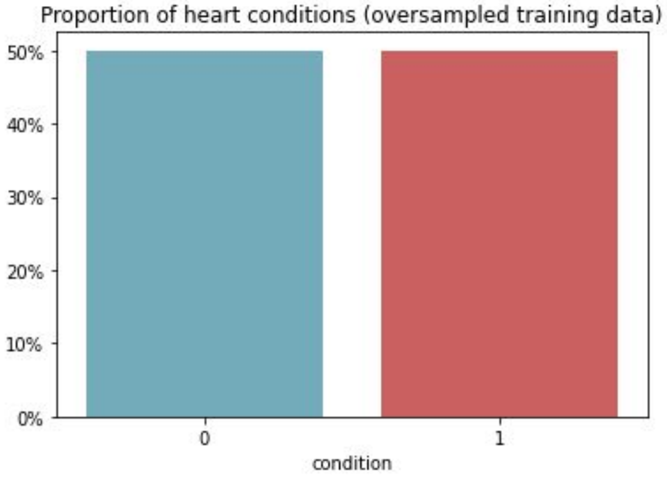
Train-Test split + SMOTE

- Unbalanced datasets are common in the medical domain
- An over-represented majority class may create bias when building a model
- Solution:
  - Oversampling (Synthesizing minority class examples)
    - Synthetic Minority Oversampling Technique (SMOTE)



SMOTE

```
oversample = SMOTE(random_state=42)
X_train_SMOTE, y_train_SMOTE = oversample.fit_resample(X_train, y_train)

oversampled_df = X_train_SMOTE.assign(condition = y_train_SMOTE)
condition_proportion(y_train_SMOTE, "Proportion of heart conditions (oversampled training data)")
print("Shape before SMOTE: " + str(X_train.shape))
print("Shape after SMOTE: " + str(X_train_SMOTE.shape))
```



Development of an algorithm: CART

CART

- Classification and Regression Trees
- Builds binary trees
- Uses a greedy approach (recursive binary splitting) to create new nodes

```python
def __init__(self, *,
             criterion = 'gini',
             max_depth = None,
             min_impurity_decrease = 0.0,
             min_impurity_split = None,
             min_samples_split = 2,
             max_features = None,
             min_samples_leaf = 1):
    self.criterion = criterion
    self.max_depth = max_depth
    self.min_impurity_decrease = min_impurity_decrease
    self.min_impurity_split = min_impurity_split
    self.min_samples_split = min_samples_split
    self.max_features = max_features
    self.min_samples_leaf = min_samples_leaf
    self.n_classes = 0
    self.root = None
```

```python
def fit(self,
        X,
        y):
    '''
    Starts the process of building a Decision Tree from a training set

    Parameters
    ----------
    X: pandas.core.frame.DataFrame, numpy.ndarray
        The training examples
    y: pandas.core.frame.DataFrame, numpy.ndarray
        The training class labels
    '''
    X, y = self.data_to_numpy(X,y)
    self.n_classes = len(np.unique(y))
    self.root = self.build_tree(X, y, self.max_depth)
```

CART - The `Partition` class

- Represents a single node in the tree
- Holds information about:
  - Left and Right child nodes
  - Feature and threshold used for splitting
  - Prediction (majority class)
  - Impurity

```
class Partition:
    def __init__(self,
                 split_feature,
                 split_value,
                 impurity_part,
                 impurity_delta,
                 impurity_feature_example,
                 prediction,
                 counter):
        self.split_feature = split_feature
        self.split_value = split_value
        self.impurity_part = impurity_part
        self.impurity_delta = impurity_delta
        self.impurity_feature_example = impurity_feature_example
        self.prediction = prediction
        self.counter = counter
        self.left_part = None
        self.right_part = None
```

CART - Building the tree (`build_tree`)

- Tree is built recursively from the top-down
- Split candidate that maximizes the reduction in impurity is selected at each step

```
split_feature, split_value, impurity_part, impurity_delta, impurity_feature_example = self.find_split(X, y)
curr_part = Partition(split_feature, split_value, impurity_part, impurity_delta, impurity_feature_example, prediction, counter)

if(split_feature is not None):
    # Temporarely create a merged df in order to filter features and classes together
    merged_df = np.append(X, np.vstack(y), axis=1)

    merged_df_left = merged_df[merged_df[:,split_feature] <= split_value]
    merged_df_right = merged_df[merged_df[:,split_feature] > split_value]

    curr_part.left_part = self.build_tree(
        merged_df_left[:, :-1],
        merged_df_left[:, -1],
        depth) # Left

    curr_part.right_part = self.build_tree(
        merged_df_right[:, :-1],
        merged_df_right[:, -1],
        depth) # Right

return curr_part
```

CART - Stopping condition

- At a certain point nodes need to be turned into leaves
  - The partition is pure
  - No possible split can be imposed
  - The constraint imposed by an hyperparameter is broken
    - Maximum depth (`max_depth`)
    - Minimum reduction in impurity (`min_impurity_decrease`)
    - Minimum impurity (`min_impurity_split`)
    - Size of partition (`min_samples_split`)
    - Size of child nodes (`min_samples_leaf`)

CART - Searching a split candidate (`find_split`)

- Nodes are analyzed in a feature-by-feature fashion

```
# Loop through all features
for feature in sampled_features:
    # Skip ahead if the feature has only one value
    if len(np.unique(X[:,feature])) == 1:
        continue

    # Sort the examples in increasing order for the selected feature
    # Consider the midpoint between two (different) adjecent values as a possible split point
    feature_sorted, y_sorted = zip(*sorted(zip(X[:,feature], y), key=itemgetter(0)))

    # Go through the sorted feature while keeping track of class occurences on each side (/partition)
    class_occ_left = Counter()
    for i in range(self.n_classes):
        class_occ_left[i] = 0 # Left partition starts as empty (all class counters are set to 0)

    class_occ_right = class_occ_part.copy() # Same as occurences of each class in the partition
```

CART - Searching a split candidate (`find_split`)

- For every feature with `k` different values, we evaluate `k-1` split thresholds
- Left & Right impurities are computed for each candidate

```
for example in range(1, size_part):
    example_class = y_sorted[example-1]

    # Increment the example class on the left, decrement it on the right
    class_occ_left[example_class] += 1
    class_occ_right[example_class] -= 1

    # Skip ahead if two adjacent values are equal
    if feature_sorted[example] == feature_sorted[example - 1]:
        continue

    size_left = example
    size_right = size_part - example

    # Skip if the threshold does not respect the min_samples_leaf parameter
    if(size_left < self.min_samples_leaf or size_right < self.min_samples_leaf):
        continue

    # Calculate the impurity of each side
    impurity_left = self.calc_impurity(class_occ_left, size_left)
    impurity_right = self.calc_impurity(class_occ_right, size_right)
```

CART - Searching a split candidate (`find_split`)

- Eventually, the split candidate that maximizes the reduction in impurity from the current partition is returned
- Split candidate = Feature + threshold

```
# Weighted sum of impurities (with selected feature at current example)
curr_impurity_feature_example = ((size_left/size_part) * impurity_left) + ((size_right/size_part) * impurity_right)

# Reduction in impurity with current split
curr_impurity_delta = impurity_part - curr_impurity_feature_example

if(self.min_impurity_decrease and curr_impurity_delta < self.min_impurity_decrease):
    continue

if(curr_impurity_delta > impurity_delta):
    split_feature = feature
    split_value = (feature_sorted[example] + feature_sorted[example-1]) / 2
    impurity_delta = curr_impurity_delta
    impurity_feature_example = curr_impurity_feature_example
```

CART - Split strategy (`calc_impurity`)

- Two metrics to evaluate the impurity of a node
  - Gini Index & Information Gain (Entropy):

$$Gini(D) = 1 - \sum_{i=1}^{C} (p_i)^2$$

$$Gini_A(D) = \frac{size(D_1)}{size(D)} Gini(D_1) + \frac{size(D_2)}{size(D)} Gini(D_2)$$

$$\Delta Gini(A) = Gini(D) - Gini_A(D)$$

$$Entropy(D) = -\sum_{i=1}^{C} p_i log_2(p_i)$$

$$Entropy_A(D) = \frac{size(D_1)}{size(D)} Entropy(D_1) + \frac{size(D_2)}{size(D)} Entropy(D_2)$$

$$Gain(A) = Entropy(D) - Entropy_A(D)$$

```python
def calc_impurity(self,
                  class_occ,
                  size_part):
    if self.criterion == 'gini':
        return (1 - sum((class_occ[curr_class]/size_part)**2 for curr_class in class_occ))
    elif self.criterion == 'entropy':
        ret = 0
        for curr_class in class_occ:
            if class_occ[curr_class]/size_part > 0:
                ret -= class_occ[curr_class]/size_part * math.log(class_occ[curr_class]/size_part, 2)
        return ret
```

CART - Pruning

- Trees are low-bias, high-variance models
- Pruning helps the model to better generalize to unseen data
  - Pre-pruning:
    - Stops the creation of some branches when the model is being built.
    - Controlled by parameters such as: `max_depth`, `min_impurity_decrease`, `min_impurity_split`, `min_samples_split` and `min_samples_leaf`
  - Post-pruning:
    - Removes sub-trees after the model has been built
    - Cost complexity pruning

CART - Predicting a class (`predict`)

- Traverse the tree from root to leaf
- Use `split_feature` and `split_value` to decide whether to go left of right
- Use the `prediction` value at leaf to predict the class label

```
def predict(self,
            X):
    X = self.data_to_numpy(X)
    y = []

    for example in X:
        curr_part = self.root

        while curr_part.right_part:
            if example[curr_part.split_feature] <= curr_part.split_value:
                curr_part = curr_part.left_part
            else:
                curr_part = curr_part.right_part
        y.append(curr_part.prediction)

    return np.array(y)
```

Development of an algorithm: Random Forest

Random Forest

- Uses the CART implementation to perform ensemble learning
- Addresses the high-variance of trees with feature randomness and bagging

```python
def fit(self,
        X,
        y):

    X, y = self.data_to_numpy(X, y)

    # Flush old forest
    if len(self.forest) > 0:
        self.forest = []

    self.n_classes = len(np.unique(y))

    for i in range(self.n_estimators):
        curr_tree = CustomDecisionTreeClassifier(criterion = self.criterion,
                                                 max_depth = self.max_depth,
                                                 min_impurity_decrease = self.min_impurity_decrease,
                                                 min_impurity_split = self.min_impurity_split,
                                                 min_samples_split = self.min_samples_split,
                                                 max_features = self.max_features,
                                                 min_samples_leaf = self.min_samples_leaf)

        if self.bootstrap:
            curr_X, curr_y = self.bootstrap_sample(X, y, self.max_samples)
            curr_tree.fit(curr_X, curr_y)
        else:
            curr_tree.fit(X, y)

        self.forest.append(curr_tree)
```

RF - Feature Randomness

- The search for a split candidate is limited to a random subset of features
- Feature sampling is done at the node level, NOT at the tree level

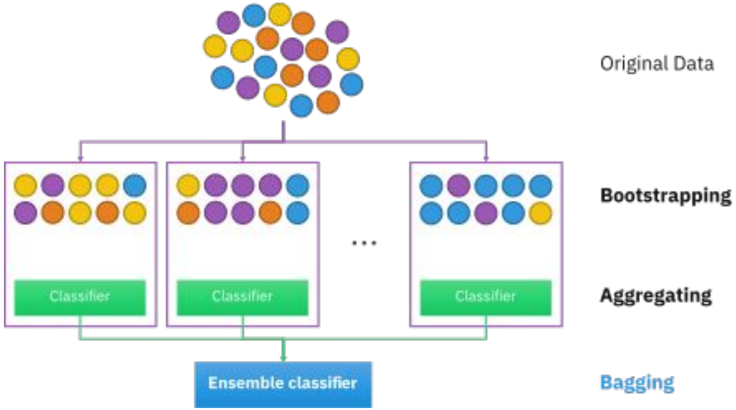
```
curr_tree = CustomDecisionTreeClassifier(criterion = self.criterion,
                                         max_depth = self.max_depth,
                                         min_impurity_decrease = self.min_impurity_decrease,
                                         min_impurity_split = self.min_impurity_split,
                                         min_samples_split = self.min_samples_split,
                                         max_features = self.max_features,
                                         min_samples_leaf = self.min_samples_leaf)
```

```
if(self.max_features == 'sqrt'):
    n=round(math.sqrt(X_cols))
elif(self.max_features == 'log2'):
    n=round(math.log2(X_cols))
elif((self.max_features is not None) and (self.max_features <= X_cols)):
    n=self.max_features

sampled_features = sorted(random.sample(range(0, X_cols), n))
```

RF - Bagging (`bootstrap_sample`)

- Bagging = Bootstrap Aggregation
- Trees in the forest are built on slightly different datasets
- Obtained by performing sampling with replacement on the original data

```python
def bootstrap_sample(X,
                     y,
                     ratio = 1):
    # Temporarily create a merged df to sample from
    merged_df = np.append(X, np.vstack(y), axis=1)

    samples_idx = np.random.choice(merged_df.shape[0], round(merged_df.shape[0]*ratio))
    merged_df[samples_idx]

    X_bootstrap = merged_df[:, :-1]
    y_bootstrap = merged_df[:, -1]

    return X_bootstrap, y_bootstrap
```



RF - Predicting a class (`predict`)

- Run the prediction on all trees in the forest
- Majority voting across all trees determines the prediction of the ensemble

```python
def predict(self,
            X):
    X = self.data_to_numpy(X)
    y_forest = [] # Each index is the list of predictions from one tree

    for tree in self.forest:
        y_forest.append(tree.predict(X))

    # Zip together prediction for same entry over different trees, extract most common class (majority voting)
    return np.array(list(map(lambda i: Counter(i).most_common(1)[0][0], zip(*y_forest))))
```

Building the models

Preparation

- Scoring function

```
def score_model(*,
                model,
                X,
                y,
                library_name,
                algorithm_name,
                fit_time,
                pruned = False,
                verbose = True,
                add_to_df = True):
```

- Hyperparameter tuning function

```
def get_best_estimator(*,
                       _param_grid,
                       _estimator,
                       X,
                       y,
                       _cv = 3,
                       _scoring = 'f1',
                       verbose = True):
```

Establishing a baseline (Sklearn implementation)

```
sklearn_tree_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': range(1,25),
    'min_impurity_decrease': [0.0, 0.1, 0.2],
    'min_samples_leaf': range(1,10),
    'min_samples_split': range(2,10)
}

sklearn_tree_params = get_best_estimator(_param_grid = sklearn_tree_grid,
                                         _estimator = DecisionTreeClassifier(random_state=42),
                                         X = X_train_SMOTE,
                                         y = y_train_SMOTE.to_numpy().flatten())

start_time = time.time()
sklearn_tree = DecisionTreeClassifier(**sklearn_tree_params, random_state=42)
sklearn_tree.fit(X_train_SMOTE, y_train_SMOTE.to_numpy().flatten())
end_time = time.time() - start_time

score_model(model = sklearn_tree,
            X = X_test,
            y = y_test,
            library_name = 'Sklearn',
            algorithm_name = 'CART',
            fit_time = end_time)
```

```
sklearn_forest_grid = {
    'max_features': range(4,21),
    'n_estimators': [100, 200, 300, 400]
}

sklearn_forest_params = get_best_estimator(_param_grid = sklearn_forest_grid,
                                           _estimator = RandomForestClassifier(random_state=42),
                                           X = X_train_SMOTE,
                                           y = y_train_SMOTE.to_numpy().flatten(),
                                           _scoring='roc_auc')

start_time = time.time()
sklearn_forest = RandomForestClassifier(**sklearn_forest_params, random_state=42)
sklearn_forest.fit(X_train_SMOTE, y_train_SMOTE.to_numpy().flatten())
end_time = time.time() - start_time

score_model(model = sklearn_forest,
            X = X_test,
            y = y_test,
            library_name = 'Sklearn',
            algorithm_name = 'Random Forest',
            fit_time = end_time)
```

Modelling using our custom implementations

```
start_time = time.time()
custom_tree = CustomDecisionTreeClassifier(**sklearn_tree_params)
custom_tree.fit(X_train_SMOTE, y_train_SMOTE)
end_time = time.time() - start_time

score_model(model = custom_tree,
            X = X_test,
            y = y_test,
            library_name = 'Custom',
            algorithm_name = 'CART',
            fit_time = end_time)
```

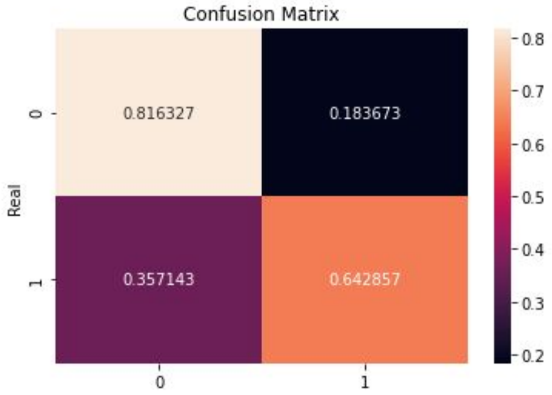
```
start_time = time.time()
custom_forest = CustomRandomForestClassifier(**sklearn_forest_params)
custom_forest.fit(X_train_SMOTE, y_train_SMOTE)
end_time = time.time() - start_time

score_model(model = custom_forest,
            X = X_test,
            y = y_test,
            library_name = 'Custom',
            algorithm_name = 'Random Forest',
            fit_time = end_time)
```

Results

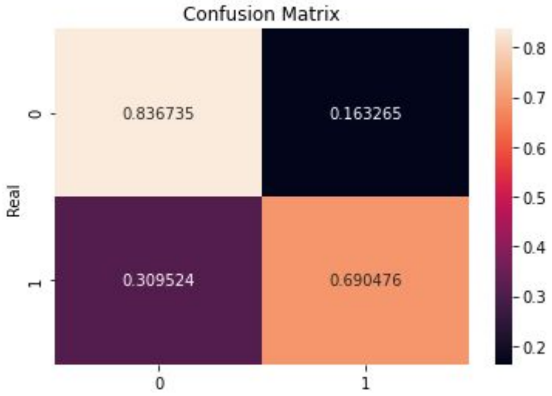
Decision Tree Classifier

- Sklearn vs Custom implementation

	precision	recall	f1-score
0	0.73	0.82	0.77
1	0.75	0.64	0.69
accuracy			0.74
macro avg	0.74	0.73	0.73
weighted avg	0.74	0.74	0.73



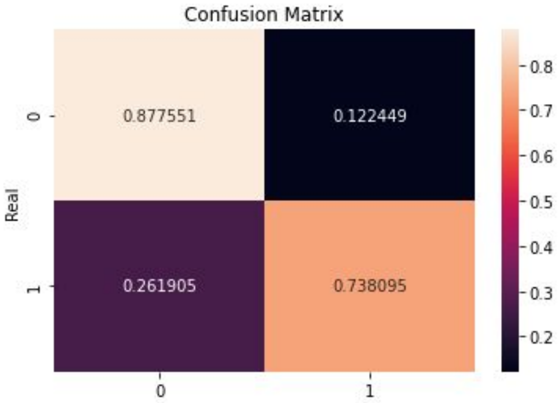
	precision	recall	f1-score
0	0.76	0.84	0.80
1	0.78	0.69	0.73
accuracy			0.77
macro avg	0.77	0.76	0.77
weighted avg	0.77	0.77	0.77



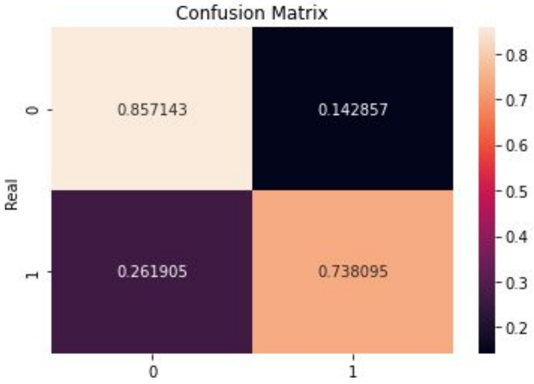
Random Forest Classifier

- Sklearn vs Custom implementation

	precision	recall	f1-score
0	0.80	0.88	0.83
1	0.84	0.74	0.78
accuracy			0.81
macro avg	0.82	0.81	0.81
weighted avg	0.82	0.81	0.81

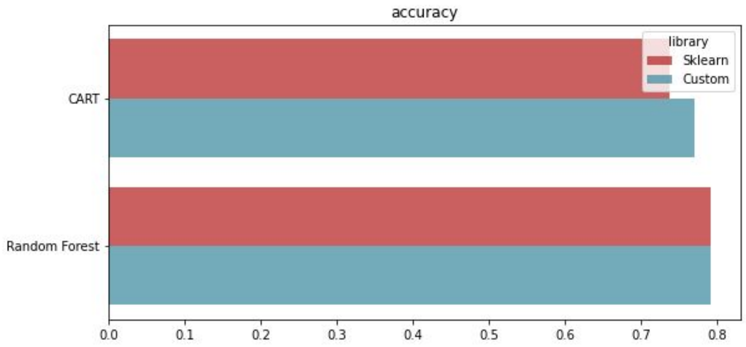


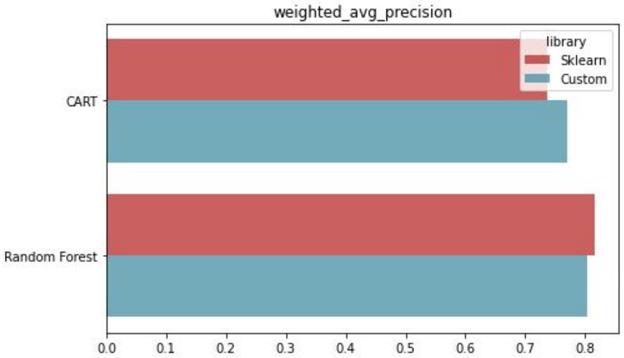
	precision	recall	f1-score
0	0.79	0.86	0.82
1	0.82	0.74	0.78
accuracy			0.80
macro avg	0.80	0.80	0.80
weighted avg	0.80	0.80	0.80

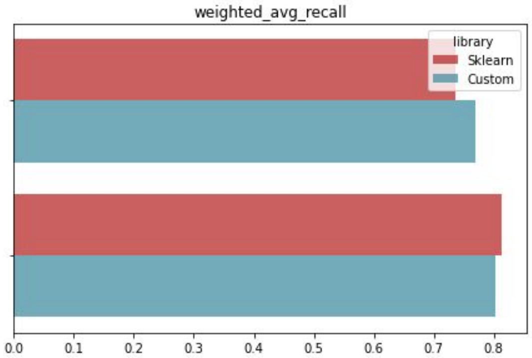


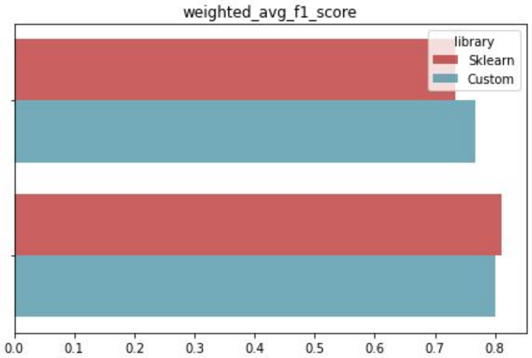
Results

- Comparable classification results across the board



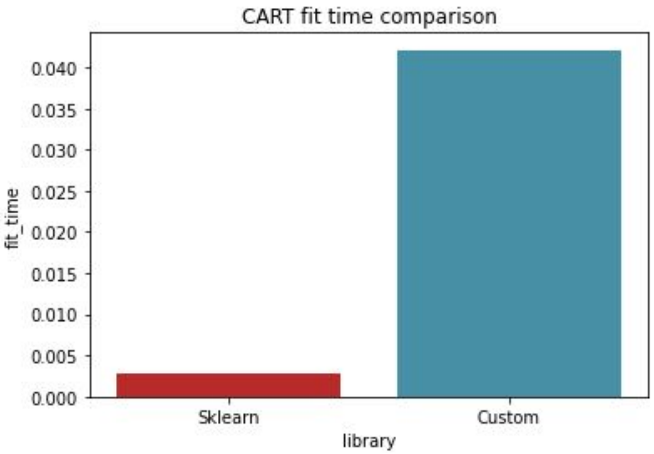


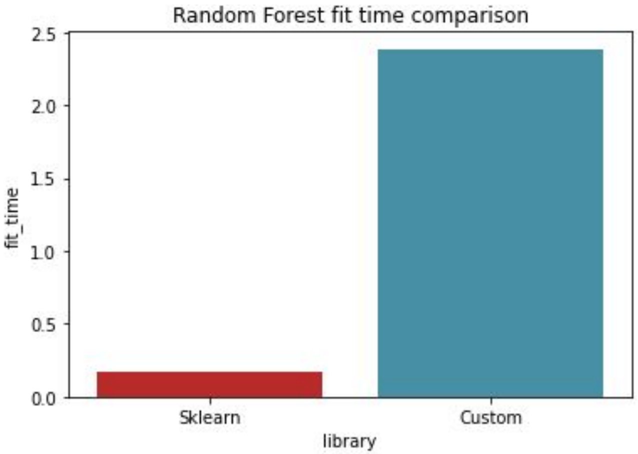




Results

- Execution time needed to build the model is higher than what is found in the Sklearn implementation





	library	algorithm	fit_time
0	Sklearn	CART	0.002825
1	Sklearn	Random Forest	0.172241
2	Custom	CART	0.042129
3	Custom	Random Forest	2.386401

Extra - CCP

Extra - Pruning with CCP

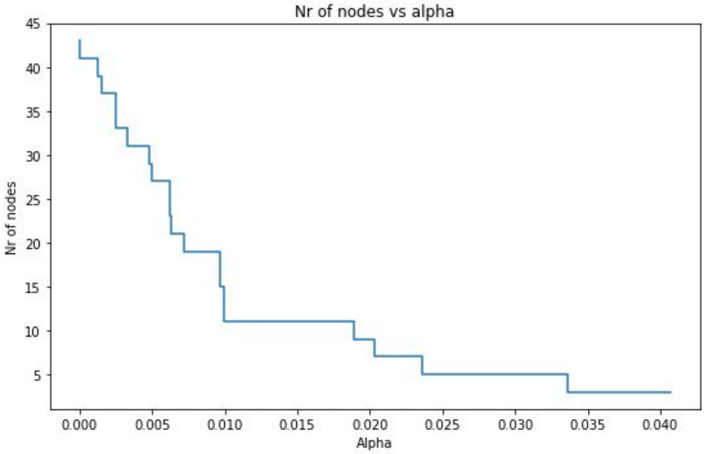
- Compute the effective alpha for all non-leaf nodes in the tree
- Nodes with the smallest alpha are the “weakest links”
    - They create trees that are redundant
    - The sub-trees they create will be pruned first

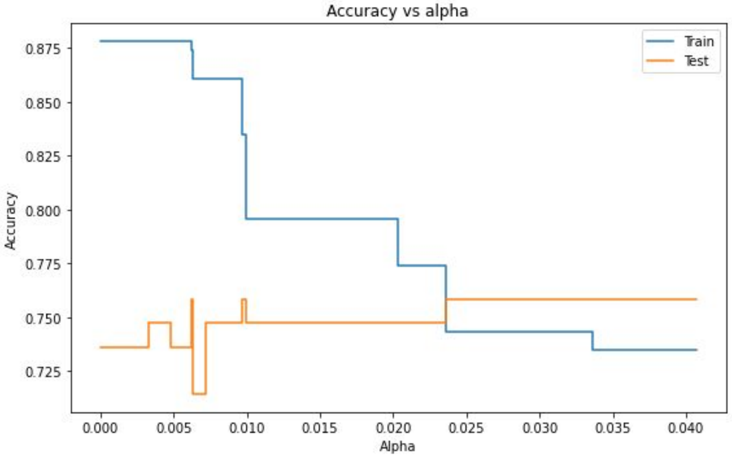
$$\alpha_{effective}(t) = \frac{R(t) - R(T_t)}{|T| - 1}$$

$$R(T_t) = \sum_{i=1}^{L} R(t_i)$$

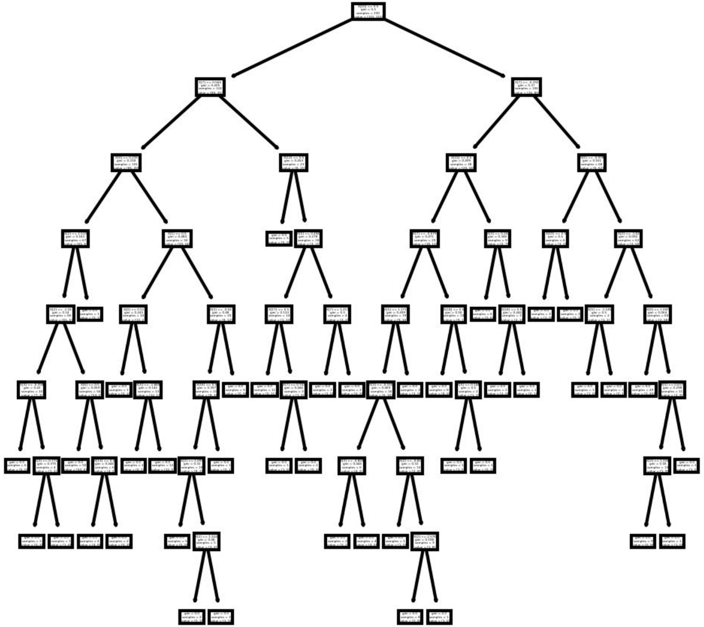
Extra - Pruning with CCP

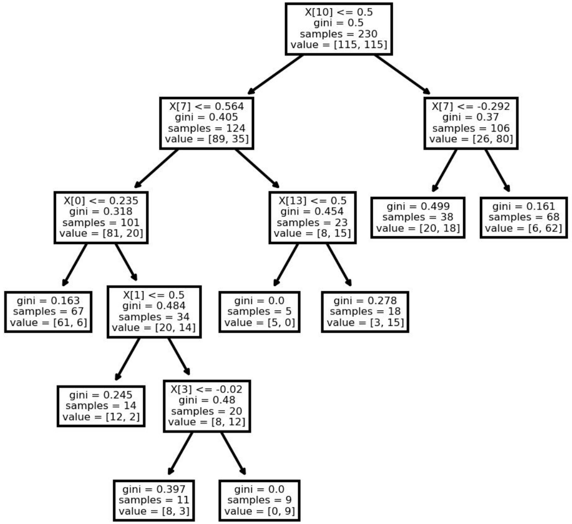
```
path = DecisionTreeClassifier(**sklearn_tree_params, random_state=42).cost_complexity_pruning_path(X_train_SMOTE, y_train_SMOTE)

# CCP alphas of subtrees + Sum of impurities of each subtree leaves
ccp_alphas, impurities = path.ccp_alphas, path.impurities

# Test the decision tree with all CCP alphas
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(**sklearn_tree_params, ccp_alpha=ccp_alpha, random_state=42)
    clf.fit(X_train_SMOTE, y_train_SMOTE)
    clfs.append(clf)

clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]
```





Extra - Pruning with CCP





Conclusions & Future Work

What did we achieve?

- Pre-processed our dataset
- Developed a custom implementation of CART and RF
- Established a baseline with the Sklearn implementation of said algorithms
- Compared the results between the two version of the algorithm
- Experimented with CCP on the Decision Tree

What could we do next?

- Look at some optimization techniques to improve fit times
- Implement CCP in our custom algorithm
- Develop a tree-based regressor

Thank you!