University of Belgrade

Faculty of Organizational Sciences

Clojure Select

Decision Support System Developed in Clojure

Mentor:

Professor Dr. Dragan Đurić

Student:

Dušan Tavić 2023/3801

Belgrade,

2024.

Table of Contents

Brief Description of the Clojure Select System

Clojure Select is a system responsible for supporting the HR sector when making decisions. The system relies on principles and methods from Decision making theory and strives to establish simple algorithms for predicting and making decisions. Within the Clojure Select system, algorithms have been implemented for predicting the success of candidates in performing their future work, which should provide decision makers in the HR sector with support in choosing the most suitable candidate for a specific job. This empowers decision-makers in the HR sector by furnishing them with robust support in selecting the most qualified candidate for a specific position.

Furthermore, the Clojure Select system implements algorithms for multi-criteria decision analysis, considering the normalization of criteria and the aggregation of their rating into a composite indicator. Notably, the system does not advocate for complete automation of the decision-making process in candidate selection - rather, it takes on the role of a supportive tool, offering informed advice grounded in mathematical principles and calculations. This approach ensures a judicious and informed decision-making environment for HR professionals.

In summary, Clojure Select stands as a powerful ally for HR professionals, integrating sophisticated algorithms to streamline decision-making. By offering insightful predictions and multi-criteria analysis, the system optimizes the candidate selection process. It serves as a valuable support tool, combining mathematical precision with human expertise to elevate decision-making in the dynamic landscape of HR.

Clojure Select Components

The Clojure Select system has three core components that collectively support the HR sector in decision-making. As shown in Figure 1, those components are:

1. Decision Tree Algorithm

2. Multi-Criteria Decision Analysis Algorithms

3. AHP (Analytic Hierarchy Process) Algorithm

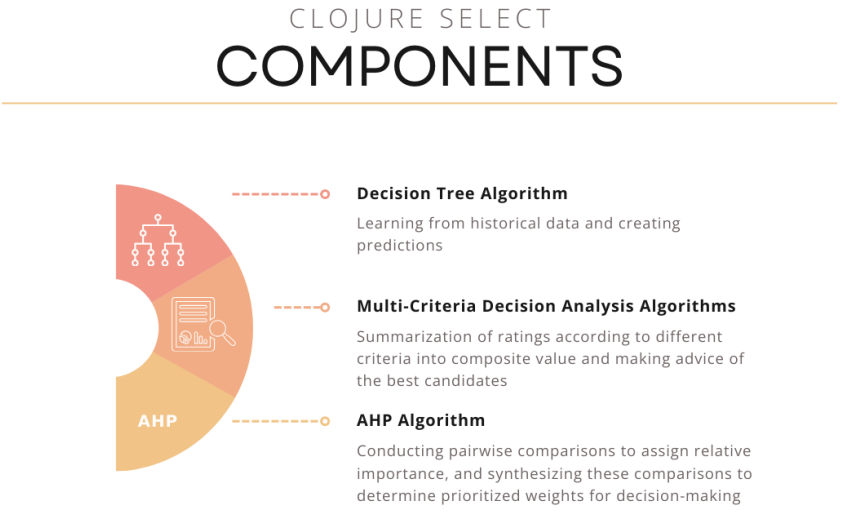

Figure 1 | Clojure Select Components

Decision Tree Algorithm

The decision tree algorithm aims to establish a tree based on historical data, modeling the way a set of input attributes translates into output attributes. A decision tree is a flowchart-like structure that represents a decision-making process or a set of decision rules. The decision tree algorithm is used for classification. This means that the main goal is to make predictions about which class of output attribute a particular entity belongs to, based on a set of input attributes characterizing that entity.

In the Clojure Select system for decision tree creation, it is necessary for the input data in the dataset to be of qualitative type. Additionally, these data should be ordinal, meaning that it should be possible to order the classes in a meaningful sequence. If the data for a specific attribute in the dataset is of numerical type, it is necessary to transform this data into ordinal data.

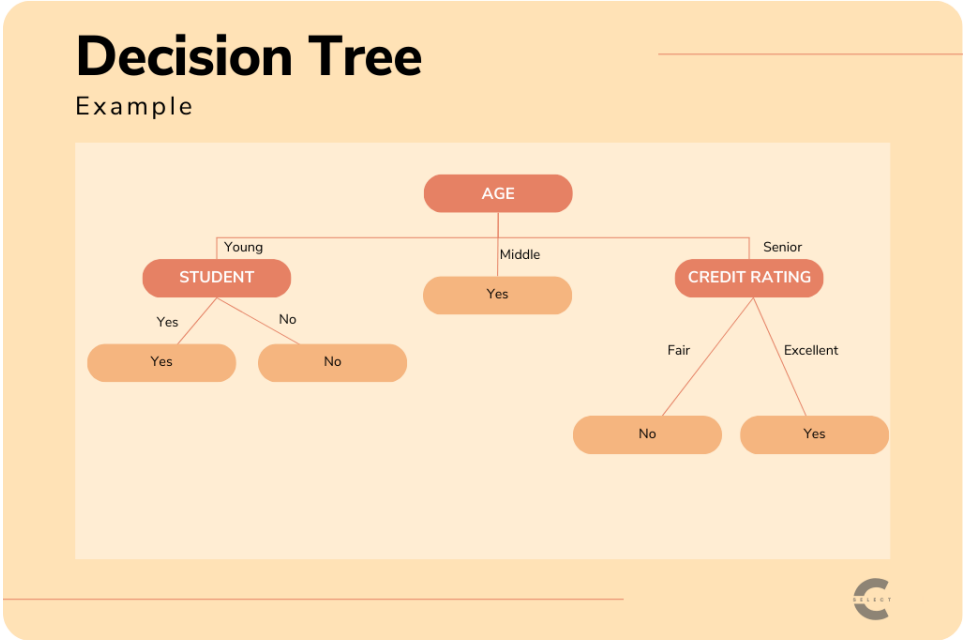

Figure 2 | Decision Tree Example

Let's assume we want to predict whether a client at a bank will be able to repay a loan. As shown in Figure 2, a decision tree based on historical data about loan repayment provides rules for decision-makers. The set of input attributes includes: Age, Student, and Credit Rating. Based on these attributes, we predict whether the client will be able to repay the loan. The attribute "Age" is depicted as the most significant for decision-making. If "Age=Young" and "Student=Yes," then the output attribute value is "Yes," indicating that the loan will be approved for the client. On the other hand, if "Age=Young" and "Student=No," the loan will not be approved for the client, as this combination of input attributes has historically often led to non-repayment. In this way, through a visually intuitive approach, we can simplify decision-making that might otherwise be very complex.

Entropy

Entropy, in the context of information theory and statistics, is a measure of uncertainty or disorder in a set of data. In information theory, it is often used to quantify the amount of information or surprise associated with the outcome of an event. Entropy quantifies uncertainty in data. We can calculate entropy using the formula shown in Formula 1.

$$H(S) = -\sum_{i=1}^{k} p_i * log_2(p_i)$$

Formula 1 | Entropy Formula

- S is the system (attribute) for which uncertainty is being calculated,
- k is the number of different values or outcomes that the attribute can have and
- pi is the probability of the occurrence of the i-th outcome.

In Clojure Select system, calculation of entropy is given by the following code:

```
(defn entropy
  "Calcuates entropy of a given column.
   As an input parameter expects only one column for which the entropy needs to be calculated."
  [data]
  (let [value-counts (frequencies data)
        values (keys value-counts)
        counts (vals value-counts)
        total-count (apply + counts)
        probabilities (map #(/ % total-count) counts)]
    (->> (map #(* % (log2 %)) probabilities)
         (reduce +)
         (* -1))))
```

Code 1 | Entropy Function

Information Gain

Information gain is a concept used in decision tree algorithms and machine learning to measure the effectiveness of a particular attribute in reducing uncertainty (entropy) about the classification of data. A higher information gain implies that using a particular attribute for splitting the data results in a more organized and less uncertain set of subsets. When splitting a decision tree, in each iteration the splitting will be done based on the attribute that currently carries the highest information gain, or in other words, based on the attribute that has the greatest significance for predictions.

In Clojure Select system, calculation of information gain is given by the following code:

```
(defn information-gain
  "Determines the significance of the input attribute in predicting the output attribute."
  [data attribute out]
  (let [total-entropy (entropy (extract-column data out))]
    (let [values-counts (frequencies (extract-column data attribute))
          values (keys values-counts)
          counts (vals values-counts)
          conditional-entropy (conditional-entropy data attribute out values counts)]
      (- total-entropy conditional-entropy))))
```

Code 2 | Information Gain Function

Function shown in Code 2 determines the significance of the input attribute in predicting the output attribute. Input parameters are: data, attribute and out. ''Data'' represents the entire training dataset that needs to be considered, ''attribute'' is the attribute for which algorithm is assessing information gain, and ''out'' is the name of the output attribute for which predictions are being made.

Conditional entropy represents the entropy for a specific subset. As shown in Formula 2, conditional entropy is the entropy of the set (attribute) S when the attribute X has the value xi.

$$H(S|X) = \sum_{i=1}^{c} p(x_i) * H(S|X = x_i)$$

Formula 2 | Conditional Entropy

In Clojure Select system, calculation of conditional entropy is given by the following code:

```
(defn conditional-entropy
  "Returns the conditional entropy for a set or subset of data."
  [data attribute out values counts]
  (reduce +
          (map (fn [i]
                 (let [value (nth values i)
                       subset (->> data
                                   (filter #(= value (get % attribute))))
                       subset-entropy (entropy (extract-column subset out))]
                   (* (/ (nth counts i) (apply + counts)) subset-entropy)))
               (range (count values)))))
```

Code 3 | Conditional Entropy Function

ID3 Algorithm

ID3 algorithm is a classic algorithm for constructing decision trees. The ID3 algorithm follows a top-down, recursive approach to build a decision tree from a set of training data.

Let's take a close look at the steps of the ID3 algorithm.

1. Calculate the entropy of the entire dataset (output attribute),

2. Choose the attribute that has the highest information gain,

3. Split the dataset into subsets based on the values of the chosen attribute,

4. Repeat steps 1-3 until:

a) the entropy of all subsets is 0, or

b) there are no more attributes for splitting, or

c) there is no attribute that reduces the entropy of the subset.

In Clojure Select system, implementation of ID3 algorithm is given by the following code:

```
(defn create-tree
  "Creates a decision tree according to training data using ID3 algorithm."
  [data out attributes]
  (if (= 1 (count (distinct (extract-column data out))))
    (first (distinct (extract-column data out)))
    (if (empty? attributes)
      (let [unique-target-values (map #(get % out) data)
            counts (frequencies unique-target-values)
            majority-class (->> counts
                                (apply max-key second)
                                first)]
        majority-class)
      (let [best-attribute (best-attribute data out attributes)]
        (if (= best-attribute nil)
          ;ako ne radimo dalje grananje, onda racunamo verovatnoce
          (let [unique-target-values (map #(get % out) data)
                counts (frequencies unique-target-values)
                majority-class (->> counts
                                    (apply max-key second)
                                    first)]
            majority-class)
          (let [attributes (remove #(= % best-attribute) attributes)]
            (->> (distinct (map #(get % best-attribute) data))
                 (reduce (fn [tree value]
                           (let [sub-data (filter #(= value (get % best-attribute)) data)
                                 subtree (create-tree sub-data out attributes)]
                             (assoc-in tree [best-attribute value] subtree)))
                         {best-attribute {}})))))))
```

Code 4 | Create Tree Function

Best attribute in each iteration is calculated by the following code:

```
(defn best-attribute
  "Returns the attribute that has the highest significance for making predictions.
   If no attribute provides supplementary information for prediction returns nil."
  [data out attributes]
  (let [information-gains (map #(information-gain data % out) attributes)
        best-attribute-index (index-of-max information-gains)
        best-attribute (nth attributes best-attribute-index)]
    (if (every? (fn [x] (= x 0.0)) information-gains)
      nil
      best-attribute)))
```

Code 5 | Best Attribute Function

Best-attribute function returns the attribute that has the highest significance for making predictions. If no attribute provides supplementary information for prediction returns nil.

Based on the created tree, the function tree-predict generates predictions for the output attribute for a new entity, while the function tree-predict-many generates predictions for the entire test dataset.

```
(defn tree-predict
  "Returns the prediction of an output variable for the entered entity, using created decision tree.
   If return value is nil, the tree cannot make prediction because the probabilies of all outcomes are equal."
  [tree entity]
  (if (map? tree)
    (let [attribute (first (keys tree))
          attribute-value (get entity attribute)]
      (tree-predict (get (get tree attribute) attribute-value) entity))
    tree))

(defn tree-predict-many
  "Returns the prediction of an output variable for all entities in entered array, using created decision tree.
   If value of output attribute is nil, the tree cannot make prediction because the probabilies of all outcomes are equal."
  [tree entities]
  (into [] (map (fn [row] (assoc row :job-fit (tree-predict tree row))) entities)))
```

Code 6 | Tree Predict and Tree Predict Many Functions

Decision Tree for Candidates in Selection Process

Let's assume we have a certain training dataset. In Figure 3, a portion of this data is shown. Complete training dataset contains information about 1018 former candidates. Note that all the data is of qualitative ordinal type.

Education	Work-experience	Technical-skills	Soft-skills	References	Communication-skills	Problem-solving-ability	Cultural-fit	Learning-ability	Job-fit
Postgraduate Education	Medium	Beginner	Low	Yes	Good	High	Moderate Fit	Limited	Low Fit
High School	Beginner	Intermediate	Medium	Yes	Needs Improvement	Low	High Fit	High	Low Fit
Postgraduate Education	Medium	Advanced	High	Yes	Good	Low	Moderate Fit	Medium	High Fit
High School	Beginner	Intermediate	High	No	Excellent	Low	High Fit	High	Good Fit
Postgraduate Education	Senior	Intermediate	High	Yes	Needs Improvement	Low	Low Fit	High	Good Fit
High School	Medium	Advanced	Low	Yes	Good	Low	Moderate Fit	Limited	Low Fit
Postgraduate Education	Medium	Advanced	High	No	Good	Medium	Low Fit	High	High Fit
Bachelor's Degree	Medium	Intermediate	Medium	Yes	Needs Improvement	High	High Fit	High	High Fit
Postgraduate Education	Beginner	Beginner	Medium	No	Needs Improvement	Medium	Low Fit	Limited	Low Fit
Postgraduate Education	Beginner	Beginner	Medium	No	Excellent	Medium	Moderate Fit	Medium	Low Fit
High School	Beginner	Advanced	High	No	Needs Improvement	Medium	Low Fit	Medium	Low Fit
Postgraduate Education	Beginner	Intermediate	Medium	Yes	Good	Low	Moderate Fit	Limited	Low Fit
Bachelor's Degree	Beginner	Beginner	Low	No	Good	High	High Fit	Limited	Low Fit
High School	Beginner	Advanced	High	No	Excellent	Medium	Low Fit	Medium	Low Fit
Bachelor's Degree	Beginner	Advanced	High	Yes	Excellent	High	High Fit	Limited	High Fit
Bachelor's Degree	Medium	Advanced	Low	Yes	Excellent	Low	High Fit	High	High Fit
Postgraduate Education	Senior	Beginner	Medium	Yes	Good	Medium	High Fit	High	High Fit
High School	Medium	Advanced	Low	Yes	Excellent	High	Low Fit	Limited	Low Fit
Bachelor's Degree	Medium	Beginner	Medium	Yes	Needs Improvement	Medium	Low Fit	Medium	Low Fit
Bachelor's Degree	Senior	Intermediate	Low	Yes	Excellent	High	Low Fit	Medium	Good Fit
Bachelor's Degree	Beginner	Intermediate	Medium	No	Good	High	High Fit	Limited	Low Fit
High School	Beginner	Beginner	Low	No	Good	High	Low Fit	High	Low Fit
Bachelor's Degree	Medium	Beginner	Low	Yes	Good	Low	Moderate Fit	Medium	Low Fit
Postgraduate Education	Senior	Beginner	Low	Yes	Excellent	High	High Fit	High	High Fit
Postgraduate Education	Beginner	Intermediate	High	No	Good	Medium	High Fit	Medium	Good Fit
Postgraduate Education	Medium	Advanced	Medium	Yes	Good	Low	High Fit	Limited	Good Fit
Bachelor's Degree	Medium	Advanced	High	No	Needs Improvement	Medium	High Fit	Limited	Good Fit
Postgraduate Education	Senior	Intermediate	Low	No	Excellent	High	High Fit	Medium	High Fit

Figure 3 | Partial Display of Training Data

Each candidate is described by nine attributes: Education, Work Experience, Technical Skills, Soft Skills, References, Communication Skills, Problem Solving Ability, Cultural Fit and Learning Ability. After a certain period of work, HR managers evaluated each candidate based on the output attribute Job Fit. Based on this data, the decision tree algorithm should infer certain patterns in the data, determining which set of input data leads to which value of the output attribute. This is intended to help HR managers decide about future candidates for whom the output attribute's value is unknown.

In the following code the training data has been read from a CSV file. Based on loaded training data, a decision tree has been created.

```
(let [attributes [:education, :work-experience, :technical-skills, :soft-skills,
                  :references, :communication-skills, :problem-solving-ability,
                  :cultural-fit, :learning-ability]
      data (into [] (->> (load-workbook "resources/candidates.xlsx")
                         (select-sheet "candidates")
                         (select-columns {:A :education, :B :work-experience,
                                          :C :technical-skills, :D :soft-skills,
                                          :E :references, :F :communication-skills,
                                          :G :problem-solving-ability, :H :cultural-fit,
                                          :I :learning-ability, :J :job-fit})
                         rest))
      tree (create-tree data :job-fit attributes)]
  (print-tree tree 0))
```

Code 7 | Decision Tree Generation

We can use the print-tree function for a visual representation of the decision tree. Given that this tree is complex, only a portion of the tree is shown in Figure 4.

```clojure
(defn print-tree [tree depth]
  (doseq [[key value] tree]
    (if (= (type value) java.lang.String)
      (println (str (apply str (repeat depth "  ")) key ": " value))
      (println (str (apply str (repeat depth "  ")) key)))
    (when (map? value)
      (print-tree value (inc depth)))))
```

Code 8 | Print Tree Function

```
:work-experience
  Medium
    :learning-ability
      Limited
        :cultural-fit
          Moderate Fit
            :technical-skills
              Beginner
                :soft-skills
                  Low: Low Fit
                  Medium: Low Fit
                  High
                    :education
                      Bachelor's Degree: Good Fit
                      High School: Low Fit
                      Postgraduate Education: Low Fit
              Advanced
                :soft-skills
                  Low: Low Fit
                  Medium: Good Fit
                  High
                    :problem-solving-ability
                      High: Good Fit
                      Medium: Good Fit
                      Low: Low Fit
              Intermediate
                :references
                  Yes
                    :soft-skills
                      High: Good Fit
                      Low: Low Fit
                  No: Low Fit
          Low Fit
            :education
              High School: Low Fit
              Bachelor's Degree: Low Fit
```

Figure 4 | Printed Decision Tree (Partial Display)

Based on the generated decision tree, we can predict the output attribute value for candidates whose output attribute value is unknown. Note that the predicted value of Job Fit attribute for first candidate is ''High Fit'', while for second candidate it is ''Good Fit''.

```clojure
(let [attributes [:education, :work-experience, :technical-skills, :soft-skills,
                  :references, :communication-skills, :problem-solving-ability,
                  :cultural-fit, :learning-ability]
      data (into [] (->> (load-workbook "resources/candidates.xlsx")
                         (select-sheet "candidates")
                         (select-columns {:A :education, :B :work-experience,
                                          :C :technical-skills, :D :soft-skills,
                                          :E :references, :F :communication-skills,
                                          :G :problem-solving-ability, :H :cultural-fit,
                                          :I :learning-ability, :J :job-fit})
                         rest))
      tree (create-tree data :job-fit attributes)
      entity {:education "Postgraduate Education",
              :work-experience "Senior",
              :technical-skills "Intermediate",
              :soft-skills "Medium",
              :references "Yes",
              :communication-skills "Excellent",
              :problem-solving-ability "Low",
              :cultural-fit "High Fit",
              :learning-ability "High"}]
  (tree-predict tree entity)) => "High Fit"
```

Code 9 | High Fit Prediction

```
(let [attributes [:education, :work-experience, :technical-skills, :soft-skills,
                  :references, :communication-skills, :problem-solving-ability,
                  :cultural-fit, :learning-ability]
      data (into [] (->> (load-workbook "resources/candidates.xlsx")
                         (select-sheet "candidates")
                         (select-columns {:A :education, :B :work-experience,
                                          :C :technical-skills, :D :soft-skills,
                                          :E :references, :F :communication-skills,
                                          :G :problem-solving-ability, :H :cultural-fit,
                                          :I :learning-ability, :J :job-fit})
                         rest))
      tree (create-tree data :job-fit attributes)
      entity {:education "High School"
              :work-experience "Beginner",
              :technical-skills "Intermediate",
              :soft-skills "Medium",
              :references "Yes",
              :communication-skills "Excellent",
              :problem-solving-ability "Low",
              :cultural-fit "High Fit",
              :learning-ability "High"}]
  (tree-predict tree entity)) => "Good Fit"
```

Code 10 | Good Fit Prediction

By using presented algorithms, it is possible to make predictions about the success of candidates in future job performance, significantly facilitating the decision-making process in the HR sector.

Model evaluation

The most common metric for model evaluation is called accuracy. It simply represents the ratio of the number of correct predictions to the total number of observations for which predictions were made.

To measure the accuracy of the model, it is necessary to split the training data, for which we know the value of the output attribute, into training and validation data.

In Clojure Select system, splitting of training dataset is given by the following code:

```
(defn training-and-validation
  "Splits the dataset into two segments based on the provided proportion.
   If the proportion is 0.8, the training data will represent 80% of the data."
  [data proportion]
  (let [count (count data)
        index (Math/round (* proportion count))
        training-test-array (split-at index data)
        training-data (into [] (get training-test-array 0))
        validation-data (into [] (get training-test-array 1))]
    [training-data validation-data]))
```

Code 11 | Training and Validation Splitting Function

Model evaluation is calculated by the following code:

```
(defn calculate-accuracy
  "Evaluates the accuracy of predictions by comparing the predicted values with the actual values,
   returning the percentage of correct predictions"
  [actuals predictions]
  (let [correct-predictions (into [] (filter (fn [prediction]
                                               (let [actual (get actuals (.indexOf predictions prediction))] (= actual prediction))) predictions))
        total-predictions (count actuals)
        accuracy (double (/ (count correct-predictions) total-predictions))]
    accuracy))
```

Code 12 | Calculate Accuracy Function

Let’s take a look at a simpler example. Let's say we want to predict whether a bank client will be able to repay a loan. Based on historical data shown in Figure 5, we will create a decision tree.

Debts	Income	Apartment	Loan-Repayment
Critical	High	Yes	No
Critical	Medium	No	No
Critical	Low	Yes	No
Critical	High	No	No
Acceptable	High	Yes	Yes
Acceptable	Low	Yes	Yes
Acceptable	Medium	Yes	Yes
Acceptable	Medium	No	No
Good	Low	No	Yes
Good	Low	No	No
Good	Low	No	No

Figure 5 | Loan-Repayment Training Data

Each observation is described by 3 ordinal attributes: Debts, Income and Apartment. In Clojure Select system, we can generate a decision tree using this data. By calling the print-tree function, we will obtain the display of a tree shown in Figure 6.

```
:debts
  critical: no
  acceptable
    :apartment
      yes: yes
      no: no
  good: no
```

Figure 6 | Loan-Repayment Decision Tree

We can now make simple predictions for new entities, for which the value of the output attribute is unknown.

```
    tree (create-tree training-data :loan-repayment [:debts :income :apartment])
    test-entity {:debts "critical"
                 :income "high"
                 :apartment "yes"}]
(tree-predict tree test-entity)) => "no"
```

Code 13 | Loan-Repayment Prediction

Multi-Criteria Decision Analysis Algorithms

Multi-criteria decision analysis (MCDA) is a decision-making approach that involves evaluating and choosing among multiple alternatives based on a set of criteria or attributes. The decision-making process becomes significantly more complex when there are multiple criteria based on which a decision needs to be made. In multi-criteria decision analysis, each alternative is described by a set of criteria that are considered when making decisions. These criteria differ in their importance for the final decision. We refer to these importance as criteria weights or ponders. Each criterion has its ponder, describing the relative significance of that criterion compared to others. This means that the total sum of all ponders assigned to defined criteria must be equal to one. This way, each criterion will participate in the decision-making process proportionally to its actual importance for the final decision.

In the context of the selection process, the decision about the most suitable candidate will be made based on various criteria, each used to evaluate every candidate. In Figure 7, we can see the model we will use for further analysis of multi-criteria decision-making theory.

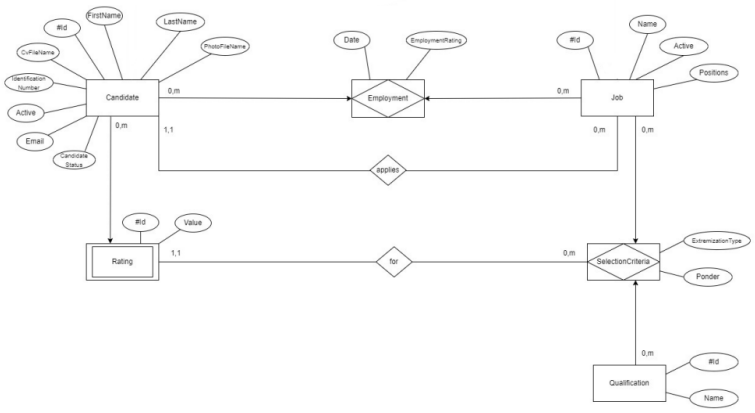

Figure 7 | Model

Normalization of Criteria

As mentioned, each candidate is described by various criteria defined for the job they are applying for. Initially, different criteria take their values from different sets that can be very diverse and disproportionate. In this case, the criteria cannot be reliably compared to each other. For example, the criterion Work Experience may take small values like 9 or 15, while the criterion Desired Salary can be expressed in hundreds of thousands, depending on the currency. It is evident that, in this case, the Desired Salary criterion will be disproportionately more significant in the decision-making process, even though it may not be the case.

Therefore, it is necessary to perform normalization for all criterion values. Through the normalization process, all values of the attributes will be scaled to the same range, typically from 0 to 1. In the Clojure Select system, for the purpose of normalization, each value is divided by the total sum of all values for a specific criterion. This way, all values will be in the range from 0 to 1, and their relative importance compared to other values of that criterion will remain unchanged.

In Clojure Select system, normalization of criteria is given by the following code:

```clojure
(defn sum-of-all-ratings
  "Calculates sum of ratings of all candidates who have applied for a specific job according to a specific criteria"
  ([job-id qualification-id]
   (reduce + (map :value (get-ratings-of-criteria job-id qualification-id))))
  ([job-id qualification-id ratings]
   (reduce + (map :value (get-ratings-of-criteria job-id qualification-id ratings)))))

(defn get-normalized-ratings-of-criteria
  "Calculates normalized values of ratings according to a specific criteria"
  ([job-id qualification-id]
   (into [] (map (fn [row] (assoc row :normalized-value (/ (double (:value row)) (sum-of-all-ratings job-id qualification-id)))) (get-ratings-of-criteria job-id qualification-id))))
  ([job-id qualification-id ratings]
   (into [] (map (fn [row] (assoc row :normalized-value (/ (double (:value row)) (sum-of-all-ratings job-id qualification-id ratings)))) (get-ratings-of-criteria job-id qualification-id ratings)))))

(defn get-normalized-rating
  "Returns a rating with added field :normalized-value"
  ([rating]
   (into {} (assoc rating :normalized-value (/ (double (:value rating)) (sum-of-all-ratings (:job-id rating) (:qualification-id rating))))))
  ([rating ratings]
   (into {} (assoc rating :normalized-value (/ (double (:value rating)) (sum-of-all-ratings (:job-id rating) (:qualification-id rating) ratings))))))

(defn normalize-job-ratings
  "Normalizes all ratings of all candidates for a specific job"
  ([job-id]
   (let [ratings (get-ratings-of-job job-id)]
     (map (fn [row] (get-normalized-rating row)) ratings)))
  ([job-id ratings]
   (let [ratings-of-job (get-ratings-of-job job-id ratings)]
     (map (fn [row] (get-normalized-rating row ratings)) ratings-of-job))))
```

Code 14 | Normalization Functions

After normalization, each candidate is evaluated with values ranging from 0 to 1 for all criteria. Therefore, we can say that the criteria are now comparable. As shown in Figure 8, each rating has its original value marked as ''value'' and its normalized value marked as ''normalized-value''.

```clojure
({:id 1, :candidate-id 1, :job-id 1, :qualification-id 1, :value 10.0, :normalized-value 0.5555555555555556}
 {:id 2, :candidate-id 1, :job-id 1, :qualification-id 2, :value 10.0, :normalized-value 0.5235602094240838}
 {:id 3, :candidate-id 1, :job-id 1, :qualification-id 3, :value 10.0, :normalized-value 0.5882352941176471}
 {:id 4, :candidate-id 2, :job-id 1, :qualification-id 1, :value 8.0, :normalized-value 0.4444444444444444}
 {:id 5, :candidate-id 2, :job-id 1, :qualification-id 2, :value 9.1, :normalized-value 0.4764397905759162}
 {:id 6, :candidate-id 2, :job-id 1, :qualification-id 3, :value 7.0, :normalized-value 0.4117647058823529})
```

Figure 8 | Normalized Values of Criteria

Aggregation

After criteria values are normalized, they become comparable. As mentioned, in multi-criteria decision analysis, there is often inequality in the importance of criteria for making the final decision. Certain criteria are highly significant, while others are desirable but not decisive. For this reason, each criterion is assigned a weight, indicating its relative importance compared to all other criteria.

Through the aggregation process, each alternative receives its final value that accumulates all information about the candidate and the importance of criteria, often referred to as a composite indicator. All criteria are aggregated into a single final criterion, thereby simplifying the multi-criteria decision-making process into a straightforward single-criterion decision process.

In the Clojure Select system, aggregation is performed using the weighted sum method. This means that all normalized values are multiplied by the relative significance of their criterion, and then the values obtained in this way for each alternative across all criteria are summed, yielding a composite aggregated rating for the candidate.

```clojure
(defn add-ponder-to-normalized-ratings
  "Adds a ponder to the normalized ratings of all candidates who applied for a specific job"
  ([job-id]
   (let [normalized-ratings (normalize-job-ratings job-id)]
     (into [] (map (fn [row] (assoc row :ponder (get-ponder job-id (:qualification-id row)))) normalized-ratings))))
  ([job-id ratings criteria]
   (let [normalized-ratings (normalize-job-ratings job-id ratings)]
     (into [] (map (fn [row] (assoc row :ponder (get-ponder job-id (:qualification-id row) criteria))) normalized-ratings)))))

(defn aggregate-candidate
  "Aggregates the overall rating of the candidate by using a weighted sum"
  ([candidate-id]
   (let [candidate (get-candidate candidate-id)
         ratings (add-ponder-to-normalized-ratings (:job-id candidate))]
     (assoc candidate :final-score (double (reduce + (map (fn [row] (* (:normalized-value row) (:ponder row))) (filter (fn [rating] (= (:candidate-id rating) candidate-id)) ratings)))))))
  ([candidate-id candidates ratings criteria]
   (let [candidate (get-candidate candidate-id candidates)
         normalized-ratings (add-ponder-to-normalized-ratings (:job-id candidate) ratings criteria)]
     (assoc candidate :final-score (double (reduce + (map (fn [row] (* (:normalized-value row) (:ponder row))) (filter (fn [rating] (= (:candidate-id rating) candidate-id)) normalized-ratings))))))))
```

Code 15 | Aggregation Functions

This way, each candidate receives their final aggregated ratings, as shown in Figure 9.

```
{:id 1,
 :firstname "Dusan",
 :lastname "Tavic",
 :active true,
 :email "dusantavic1@gmail.com",
 :status 0,
 :job-id 1,
 :final-score 0.5524928994285323}
clj:clojureselect.business-logic:>
{:id 2,
 :firstname "Nenad",
 :lastname "Panovic",
 :active true,
 :email "nenadpann@gmail.com",
 :status 0,
 :job-id 1,
 :final-score 0.44750710057146764}
```

Figure 9 | Final Score of Candidates

Decision Support

Let's assume that in the relational database, we have the following values stored in the Ratings table:

candidate-id	job-id	qualification-id	value
1	1	1	7
1	1	2	8.1
1	1	3	7.8
2	1	1	9.5
2	1	2	10
2	1	3	6.1
3	1	1	7.8
3	1	2	7.9
3	1	3	9.5

Figure 10 | Ratings in Database

Based on these values, it is necessary to first perform the normalization of criteria, so that all criteria take values from the same range. After that, it is required to aggregate all ratings into a composite indicator for each candidate. Note that the ponders of criteria are also stored in the database in the Criteria table.

In the Clojure Select system, the decision-support function is used to recommend the most suitable candidates for a job. This function implements all the principles we have mentioned so far.

```
(defn decision-support
  "Applies the method of multi-criteria decision-making and provides
   advices for the most suitable candidates for a specific job"
  ([job-id]
   (let [candidates (get-candidates job-id)]
     (into [] (sort-by :final-score (comparator >) (map (fn [row] (aggregate-candidate (:id row))) candidates)))))
  ([job-id candidates ratings criteria]
   (let [candidates-for-job (get-candidates job-id candidates)]
     (into [] (sort-by :final-score (comparator >) (map (fn [row] (aggregate-candidate (:id row) candidates ratings criteria)) candidates-for-job))))))
```

Code 16 | Decision Support Function

The decision-support function will return a sorted list of candidates with final ratings, which we have called the composite indicators. The candidates in the returned list are sorted from the most suitable to the least suitable for the specific job.

```
[{:id 2,
  :firstname "Dragana",
  :lastname "Mirkovic",
  :active true,
  :email "gaga@gmail.com",
  :status "rated",
  :job-id 1,
  :final-score 0.3629946185501741}
 {:id 3,
  :firstname "Maja",
  :lastname "Petrovic",
  :active true,
  :email "mayapetrovic@gmail.com",
  :status "unrated",
  :job-id 1,
  :final-score 0.33284425451092114}
 {:id 1,
  :firstname "Marko",
  :lastname "Radovic",
  :active true,
  :email "marko@gmail.com",
  :status "rated",
  :job-id 1,
  :final-score 0.30416112693890474}]
```

Figure 11 | Decision Support Result

In addition, we can use the selection-advice function for a more comprehensible presentation of the recommendations for the most suitable candidates.

```
clj:clojureselect.core-test:>
["Top rated Candidate for C# Junior Developer: Dragana Mirkovic with final score of 0.363"
 "You should also consider Maja Petrovic with final score of 0.3328 for C# Junior Developer"]
```

Figure 12 | Selection Advice

AHP Algorithm

In decision-making theory AHP stans for Analytic Hierarchy Process. It is a decision-making technique that helps individuals and groups make complex decisions by structuring them into a hierarchical model. AHP provides a structured and systematic approach to decision-making, particularly in situations where multiple criteria and stakeholders are involved. In multi-criteria decision analysis, the AHP algorithm can be used to determine the significance of criteria. The decision-makers uses Saaty's 9-point scale to indicate how important one criterion is relative to another, employing pairwise comparisons.

Let's assume we have three criteria for a specific job: Education, Work Experience and Cultural Fit. Decision-makers assess the importance of each criterion relative to another, and these values are entered into AHP matrix, as shown in Figure 13.

	Education	Work Experience	Cultural Fit
Education		3	2
Work Experience			0.5
Cultural Fit			

Figure 13 | Initial AHP Matrix

A characteristic of the AHP matrix is that inverse values are symmetrically located across the diagonal. If criterion A is 3 times more significant than criterion B, then criterion B is 1/3 as significant as criterion A. Additionally, the values on the main diagonal are always equal to 1 because they represent the comparison of a criterion with itself.

In the Clojure Select system, the functions responsible for creating the AHP matrix are shown in the following code:

```
(defn inverse-ponders
  "Inverts AHP ponders. If significance value on a specific position is x, then significance value on inverted position must be 1/x"
  [raw-ponders]
  (into [] (map (fn [obj] (let [position1 (get (:position obj) 0)
                                position2 (get (:position obj) 1)] (assoc obj :position [position2, position1] :significance (/ 1 (:significance obj))))) raw-ponders)))

(defn add-inverse-ponders
  "Adds inverted ponders to an array of initially created ponders"
  [raw-ponders]
  (into [] (concat raw-ponders (inverse-ponders raw-ponders))))

(defn get-ahp-ponders
  "Returns ahp ponders for a specific job"
  [job-id ahp-ponders]
  (into [] (filter (fn [row] (= (:job-id row) job-id)) ahp-ponders)))


(defn create-matrix
  "Creates a matrix with specific number of rows and columns, initially filled with ones"
  ([rows columns]
   (vec (for [x (range rows)]
          (vec (repeat columns 1)))))
  ([rows-and-cols]
   (vec (for [x (range rows-and-cols)]
          (vec (repeat rows-and-cols 1))))))

(defn modify-ahp-matrix
  "Recursive function that modifies values in ahp matrix"
  [matrix ponders]
  (if-not (= (rest ponders) [])
    (let [current-ponder (first ponders)
          mat (modify-ahp-matrix matrix (into [] (rest ponders)))]
      (assoc-in mat [(get (:position current-ponder) 0) (get (:position current-ponder) 1)] (:significance current-ponder)))
    (let [current-ponder (first ponders)]
      (assoc-in matrix [(get (:position current-ponder) 0) (get (:position current-ponder) 1)] (:significance current-ponder)))))
```

Code 17 | AHP Matrix Help-Functions

```clojure
(defn create-ahp-matrix
  "Creates AHP matrix of assessments for a specific job"
  [job-id ahp-ponders]
  (let [raw-ponders (get-ahp-ponders job-id ahp-ponders)
        final-ponders (add-inverse-ponders raw-ponders)
        rows-cols-count (count ahp-ponders)
        init-matrix (create-matrix rows-cols-count)]
    (modify-ahp-matrix init-matrix final-ponders)))
```

Code 18 | Create AHP Matrix Function

After calling the create-ahp-matrix function, the Clojure Select system will create the matrix shown in Figure 14.

```
clj:clojureselect.core-test:>
1 3 2
1/3 1 0.5
1/2 2.0 1
```

Figure 14 |Created AHP Matrix

The obtained matrix will be used to calculate the importance of each criterion. The AHP method is shown in Figure 15. In the figure, we see that the AHP method involves summing the values in each row of the matrix. These values are called initial sums. After that, each initial sum is divided by the total value of all initial sums to calculate their relative values, which will sum up to one. These values obtained in this way represent the final ponders for each criterion.

	Education	Work Experience	Cultural Fit		Init. sum	Ponder
Education	1	3	2		6	0.529
Work Experience	0.3333	1	0.5		1.8333	0.162
Cultural Fit	0.5	2	1		3.5	0.309
					Sum of Init. sums	
					11.333	

Figure 15 | AHP Calculation Method

In the Clojure Select system, AHP calculation is given by the following code:

```
(defn calculate-total-array
  "Calculates total array of weights for all criteria using AHP methodology"
  [job-id ahp-ponders]
  (let [ahp-matrix (create-ahp-matrix job-id ahp-ponders)
        array-sums (into [] (map (fn [arr] (reduce + arr)) ahp-matrix))
        total-sum (reduce + array-sums)]
    (into [] (map (fn [element] (/ element total-sum)) array-sums))))

(defn calculate-ahp
  "Calculates total ahp weights for criteria"
  [job-id ahp-ponders]
  (let [ahp-array (calculate-total-array job-id ahp-ponders)
        criteria (get-jobs-criteria job-id)]
    (into [] (map (fn [element] (assoc element :ahp-ponder (get ahp-array (.indexOf criteria element)))) criteria))))
```

Code 19 | AHP Calculation Function

After calling calculate-ahp function, the Clojure Select system will return the criteria with new ponders obtained through the mentioned AHP method, as shown in Figure 16.

```
clj:clojureselect.core-test:>
[{:job-id 1, :qualification-id 1, :ponder 0.5, :ahp-ponder 0.5294117647058824}
 {:job-id 1, :qualification-id 2, :ponder 0.3, :ahp-ponder 0.16176470588235292}
 {:job-id 1, :qualification-id 3, :ponder 0.2, :ahp-ponder 0.3088235294117647}]
```

Figure 16 | Calculated AHP Ponders

Literature

Barzilai, J., & Golany, B. (1994). AHP rank reversal, normalization and aggregation rules. INFOR: Information Systems and Operational Research, 32(2), 57-64.

Charbuty, B., & Abdulazeez, A. (2021). Classification based on decision tree algorithm for machine learning. Journal of Applied Science and Technology Trends, 2(01), 20-28.

Edwards, W. (1954). The theory of decision making. Psychological bulletin, 51(4), 380.

Fitz-Enz, J. (2010). The new HR analytics. American Management Association.

Ihsan, Z., Idris, M. Y., & Abdullah, A. H. (2013). Attribute normalization techniques and performance of intrusion classifiers: A comparative analysis. Life Science Journal, 10(4), 2568-2576.

Majumder, M., & Majumder, M. (2015). Multi criteria decision making. Impact of urbanization on water shortage in face of climatic aberrations, 35-47.

Marler, J. H., & Boudreau, J. W. (2017). An evidence-based review of HR Analytics. The International Journal of Human Resource Management, 28(1), 3-26.

Mohammed, D. A. Q. (2019). HR analytics: a modern tool in HR for predictive decision making. Journal of Management, 6(3).

Navada, A., Ansari, A. N., Patil, S., & Sonkamble, B. A. (2011, June). Overview of use of decision tree algorithms in machine learning. In 2011 IEEE control and system graduate research colloquium (pp. 37-42). IEEE.

Podvezko, V. (2009). Application of AHP technique. Journal of Business Economics and Management, (2), 181-189.

Saaty, T. L. (2003). Decision-making with the AHP: Why is the principal eigenvector necessary. European journal of operational research, 145(1), 85-91.

Saaty, T. L. (2004). Decision making—the analytic hierarchy and network processes (AHP/ANP). Journal of systems science and systems engineering, 13, 1-35.

Song, Y. Y., & Ying, L. U. (2015). Decision tree methods: applications for classification and prediction. Shanghai archives of psychiatry, 27(2), 130.

Suknović, M., Delibašić, B., Jovanović, M., Vukićević, M., & Radovanović, S. (2021). Odlučivanje, Fakultet organizacionih nauka.

Triantaphyllou, E., & Triantaphyllou, E. (2000). Multi-criteria decision making methods (pp. 5-21). Springer US.