

# ALG5I - Übung 02

Alen Kocaj

01. November 2017

## 1 Hidden Markov Models & Forward Algorithmus

**Implementieren Sie in einer Programmiersprache Ihrer Wahl ein Framework für Hidden Markov Models (HMMs).**

Diese Aufgabe wurde in Google Programmiersprache Go umgesetzt, da diese neben intuitiverer Syntax als andere Programmiersprachen auf Typsicherheit setzt. Das git repository für die Aufgabe findet sich unter [github.com/rathalos64/algo5](https://github.com/rathalos64/algo5).

Der geschätzte Arbeitsaufwand beträgt 12h.

Die Struktur der Abgabe besteht aus den folgenden Files.

```
/root
├── main.go
├── hmm.go
├── sequence.go
├── forward.go
├── structs.go
└── utils.go
```

Kurz zur Erklärung der einzelnen Files.

- **main.go** präsentiert das Hauptprogramm, welches die Models und Sequenzen einliest, validiert und anschließend evaluiert.
- **hmm.go** definiert die Struktur eines Hidden Markov Models mitsamt Constraints und Mappingnamen für Se-/Deserialisierung.
- **sequence.go** definiert die Struktur einer Sequenz von Beobachtungen mitsamt Constraints und Mappingnamen für Se-/Deserialisierung.
- **forward.go** implementiert den Forward-Algorithmus über die Funktion *Evaluate()*.
- **structs.go** definiert Hilfstypen (u.a für die Validierung).
- **utils.go** definiert Hilfsfunktionen

Das Programm arbeitet mit Inputfiles, welche zum Definieren von HMMs und Sequenzen verwendet werden. Als Fileformat wird JSON verwendet aufgrund Go's nativer Se-/Deserialisierungstechnik. Zudem sind JSON files leichter lesbar und in vielen Webanwendungen standardisiert in Verwendung. Das Programm verwendet genau ein JSON file für das Einlesen von verschiedenen Modellen [Listing 1] und eines für das Einlesen der Beobachtungssequenzen [Listing 2].

```
[
  {
    "id": "hmm_01",
    "states": ["rainy", "cloudy", "sunny"],
    "observations": ["humid", "medium", "dry"],

    "initial": {"rainy": 0.3, "cloudy": 0.5, "sunny": 0.2},
    "transitions": {
      "rainy": {"rainy": 0.4, "cloudy": 0.3, "sunny": 0.3},
      "cloudy": {"rainy": 0.2, "cloudy": 0.6, "sunny": 0.2},
      "sunny": {"rainy": 0.1, "cloudy": 0.1, "sunny": 0.8}
    },
    "emissions": {
      "rainy": {"humid": 0.4, "medium": 0.5, "dry": 0.1},
      "cloudy": {"humid": 0.3, "medium": 0.4, "dry": 0.3},
      "sunny": {"humid": 0.2, "medium": 0.2, "dry": 0.6}
    }
  },
  {
    "id": "hmm_02",
    "states": ["rainy", "cloudy", "sunny"],
    "observations": ["humid", "medium", "dry"],

    "initial": {"rainy": 0.5, "cloudy": 0.3, "sunny": 0.2},
    "transitions": {
      "rainy": {"rainy": 0.4, "cloudy": 0.3, "sunny": 0.3},
      "cloudy": {"rainy": 0.1, "cloudy": 0.8, "sunny": 0.1},
      "sunny": {"rainy": 0.5, "cloudy": 0.3, "sunny": 0.2}
    },
    "emissions": {
      "rainy": {"humid": 0.3, "medium": 0.4, "dry": 0.3},
      "cloudy": {"humid": 0.4, "medium": 0.5, "dry": 0.1},
      "sunny": {"humid": 0.2, "medium": 0.2, "dry": 0.6}
    }
  },
  {
    "id": "hmm_03",
    "states": ["rainy", "cloudy", "sunny"],
    "observations": ["humid", "medium", "dry"],

    "initial": {"rainy": 0.1, "cloudy": 0.1, "sunny": 0.8},
    "transitions": {
      "rainy": {"rainy": 0.5, "cloudy": 0.3, "sunny": 0.2},
      "cloudy": {"rainy": 0.1, "cloudy": 0.8, "sunny": 0.1},
      "sunny": {"rainy": 0.4, "cloudy": 0.3, "sunny": 0.3}
    },
    "emissions": {
      "rainy": {"humid": 0.2, "medium": 0.2, "dry": 0.6},
      "cloudy": {"humid": 0.5, "medium": 0.1, "dry": 0.4},
      "sunny": {"humid": 0.3, "medium": 0.4, "dry": 0.3}
    }
  }
]
```

]

Listing 1: models.json. Zu sehen sind hier drei Modelle, welches das Wetter simulieren sollen. Zu beachten sind die jeweils leichte Veränderung in PI, A und B. Durch diese Parameterveränderung, wollen wir später sehen, welches der Modelle am Wahrscheinlichsten Beobachtungssequenzen erzeugt haben könnten.

[

```

{
  "id": "sequence_01",
  "hmm_ids": ["hmm_01", "hmm_02", "hmm_03"],

  "observations": ["dry", "humid", "medium"]
},
{
  "id": "sequence_02",
  "hmm_ids": ["hmm_01", "hmm_02"],

  "observations": ["dry", "humid"]
},
{
  "id": "sequence_03",
  "hmm_ids": ["hmm_01"],

  "observations": ["dry", "humid", "medium", "humid", "dry"]
}

```

]

Listing 2: sequences.json. Hier zu sehen sind drei Definitionen für Beobachtungssequenzen. Die eigentliche Beobachtungssequenz wird über das Feld “observations” definiert. Das Feld “hmm\_ids” beinhaltet die IDs der Modelle, gegen diese getestet werden soll. Es ist offensichtlich, dass die IDs von Modellen stammen müssen, welche existieren. So wird ersichtlich, dass z.b.: die Sequenz “sequence\_01” gegen alle Modelle getestet wird.

Das Programm wurde für die Konsole entwickelt und verwendet daher Kommandozeilenparameter um die Inputdaten einzulesen [Listing 3].

```

$ ./hmm -models examples/models/models.json -sequences examples/sequences/
sequences.json

```

Listing 3: Beispielaufruf am Terminal

Hat das Programm erfolgreich die Inputfiles eingelesen und deren Korrektheit validiert, printet das Programm sofort das Ergebnis auf das Terminal. Es wird zu jeder Sequenz alle Wahrscheinlichkeiten (Likelihood & Loglikelihood) der jeweiligen Modelle sowie das wahrscheinlichste (likeliest) Modell ausgegeben [Abbildung 1].

```

=====
[i] Evaluating Sequence "sequence_01"
## using "hmm_01"
> L(O|M) = 0.0276120000
> log L(O|M) = -3.5895048181
-----
## using "hmm_02"
> L(O|M) = 0.0385890000
> log L(O|M) = -3.2547880172
-----
## using "hmm_03"
> L(O|M) = 0.0200720000
> log L(O|M) = -3.9084294699
-----
## [RESULT] argmax(Mj) of P("sequence_01" | Mj) is
> ID = "hmm_02"
> L(O|M) = 0.0385890000
> log L(O|M) = -3.2547880172
=====
=====
[i] Evaluating Sequence "sequence_02"
## using "hmm_01"
> L(O|M) = 0.0819000000
> log L(O|M) = -2.5022562881
-----
## using "hmm_02"
> L(O|M) = 0.0933000000
> log L(O|M) = -2.3719351711
-----
## [RESULT] argmax(Mj) of P("sequence_02" | Mj) is
> ID = "hmm_02"
> L(O|M) = 0.0933000000
> log L(O|M) = -2.3719351711
=====

```

Abbildung 1: Ein Ausschnitt des Terminals. Zu jeder Sequenz ermittelt das Programm das Modell, welches die Sequenz am Wahrscheinlichsten erzeugt haben könnte.

Listing 4: main.go

```
1 package main
2
3 import (
4     "encoding/json"
5     "flag"
6     "fmt"
7     "log"
8     "math"
9     "os"
10 )
11
12 // Application is the main application and contains
13 // all necessary information for the whole program to work.
14 // It stores all defined models and all sequences of observations.
15 type Application struct {
16     HMMs      []HMM
17     Sequences []Sequence
18 }
19
20 func main() {
21     // define command line flags
22     precision := flag.String(
23         "precision",
24         "10",
25         "the numerical precision for L(O|M) and log L(O|M)")
26     hmmPath := flag.String(
27         "models",
28         "examples/models/models.json",
29         "the path to the hmm json file")
30     sequencePath := flag.String(
31         "sequences",
32         "examples/sequences/sequences.json",
33         "the path to the sequence json file")
34     flag.Parse()
35
36     app := Application{}
37
38     // read models
39     err := app.ReadHMMs(*hmmPath)
40     if err != nil {
41         log.Fatalf("Couldn't read HMMs from %q: %s", hmmPath, err)
42     }
43
44     // validate all models
45     validationResult := app.ValidateHMMs()
46     if !validationResult.Valid() {
47         log.Fatalf("Validation of all HMMs failed: \n%s",
48             validationResult.Marshal())
49     }
50     // validate each models individually
51     for _, hmm := range app.HMMs {
52         validationResult := hmm.Validate()
53         if !validationResult.Valid() {
```

```

54         log.Fatalf("Validation of HMM %q failed: \n%s", hmm.ID,
55             validationResult.Marshal())
56     }
57 }
58
59 // read sequences
60 err = app.ReadSequences(*sequencePath)
61 if err != nil {
62     log.Fatalf("Couldn't read Sequences from %q: %s", sequencePath, err)
63 }
64
65 // validate all models
66 validationResult = app.ValidateSequences()
67 if !validationResult.Valid() {
68     log.Fatalf("Validation of all Sequences failed: \n%s",
69         validationResult.Marshal())
70 }
71 // validate each sequences individually
72 for _, sequence := range app.Sequences {
73     validationResult := sequence.Validate(app.HMMs)
74     if !validationResult.Valid() {
75         log.Fatalf("Validation of Sequence %q failed: \n%s",
76             sequence.ID, validationResult.Marshal())
77     }
78 }
79
80 // evaluate for each sequence each model
81 for _, sequence := range app.Sequences {
82     fmt.Printf("=====\n")
83     // the best model for the sequence
84     best := Result{}
85
86     fmt.Printf("[i] Evaluating Sequence %q\n", sequence.ID)
87     for _, hmm := range app.HMMs {
88         // test only against the target HMMs
89         if !isInStringSlice(hmm.ID, sequence.HMMIDs) {
90             continue
91         }
92
93         fmt.Printf("## using %q \n", hmm.ID)
94
95         // calculate the likelihood of model given sequence
96         likelihood := Evaluate(sequence, hmm)
97         result := Result{
98             ID:             hmm.ID,
99             Likelihood:        likelihood,
100             LogLikelihood:    math.Log(likelihood),
101         }
102
103         if result.Better(best) {
104             best = result
105         }
106     }
107 }

```

```

108
109         fmt.Printf("> L(O|M) = %."+precision+"f \n", result.Likelihood)
110         fmt.Printf("> log L(O|M) = %."+precision+"f \n", result.
            LogLikelihood)
111         fmt.Printf("-----\n")
112     }
113
114     fmt.Printf("## [RESULT] argmax(Mj) of P(%q | Mj) is\n", sequence.ID)
115     fmt.Printf("> ID = %q \n", best.ID)
116     fmt.Printf("> L(O|M) = %."+precision+"f \n", best.Likelihood)
117     fmt.Printf("> log L(O|M) = %."+precision+"f\n", best.LogLikelihood)
118
119     fmt.Printf("=====" +
120         "=====\n")
121 }
122 }
123
124 // ReadHMMs reads all defined HMMs from the given path
125 // and stores it in the application
126 func (app *Application) ReadHMMs(path string) error {
127     readerHmms, err := os.Open(path)
128     if err != nil {
129         return err
130     }
131     defer readerHmms.Close()
132
133     // deserialize
134     var hmms []HMM
135     err = json.NewDecoder(readerHmms).Decode(&hmms)
136     if err != nil {
137         return err
138     }
139
140     // set N and M for each model
141     for i := range hmms {
142         hmms[i].N = len(hmms[i].S)
143         hmms[i].M = len(hmms[i].V)
144     }
145
146     app.HMMs = hmms
147     return nil
148 }
149
150 // ValidateHMMs verifies the validity of the read HMMs
151 func (app Application) ValidateHMMs() ValidationResult {
152     validationResult := ValidationResult{}
153     uniqueElements := map[string]bool{}
154
155     if len(app.HMMs) == 0 {
156         validationResult.Add("HMMs", "No HMMs given or invalid json file
            structure")
157     }
158
159     // verify the uniqueness of the hmm by inspecting ID

```

```
160     for _, hmm := range app.HMMs {
161         if uniqueElements[hmm.ID] == true {
162             validationResult.Add("HMMs", "HMMs must have unique ID")
163             break
164         }
165         uniqueElements[hmm.ID] = true
166     }
167
168     return validationResult
169 }
170
171 // ReadSequences reads all defined sequences of observations from the
172 // given path and stores it in the application
173 func (app *Application) ReadSequences(path string) error {
174     sequenceReader, err := os.Open(path)
175     if err != nil {
176         return nil
177     }
178     defer sequenceReader.Close()
179
180     // deserialize
181     var sequences []Sequence
182     err = json.NewDecoder(sequenceReader).Decode(&sequences)
183     if err != nil {
184         return err
185     }
186
187     app.Sequences = sequences
188     return nil
189 }
190
191 // ValidateSequences verifies the validity of the read sequences
192 func (app Application) ValidateSequences() ValidationResult {
193     validationResult := ValidationResult{}
194     uniqueElements := map[string]bool{}
195
196     if len(app.Sequences) == 0 {
197         validationResult.Add("Sequences", "No sequences given or invalid
198             json file structure")
199     }
200
201     // verify the uniqueness of the hmm by inspecting ID
202     for _, sequence := range app.Sequences {
203         if uniqueElements[sequence.ID] == true {
204             validationResult.Add("Sequences", "Must have unique ID")
205             break
206         }
207         uniqueElements[sequence.ID] = true
208     }
209
210     return validationResult
211 }
```

Listing 4: main.go



Listing 5: hmm.go

```

1 package main
2
3 // State is the state the hidden system can enter
4 type State string
5
6 // ObservationSymbol is the observation within
7 // a sequence whose symbol is one from V (set of observation symbols).
8 type ObservationSymbol string
9
10 // HMM models a hidden markov model with all possible parameter as fields
11 // S: represents the individual states {s1, s2, ... sN}
12 // N: is the number of states
13 //
14 // V: represents the individual observation symbols (all observations types
15 //     known)
16 // M: is the number of distinct observation symbols per state
17 //
18 // PI: represents the initial state distribution (initial start vector)
19 // A: represents the state transition probability distribution
20 // B: represents the observation symbol probability distribution
21 //     (emission probabilities)
22 //
23 // In order to identify sequences by their observation symbol PI, A and B
24 // are
25 // keyed with the state / observation symbol.
26 type HMM struct {
27     ID string `json:"id"`
28     S []State `json:"states"`
29     N int
30
31     V []ObservationSymbol `json:"observations"`
32     M int
33
34     PI map[State]float64 `json:"initial"`
35     A  map[State]map[State]float64 `json:"transitions"`
36     B  map[State]map[ObservationSymbol]float64 `json:"emissions"`
37 }
38
39 // Validate verifies the constraints of a given HMM; not only
40 // mathematical but functional ones
41 func (hmm HMM) Validate() ValidationResult {
42     validationResult := ValidationResult{}
43
44     if hmm.ID == "" {
45         validationResult.Add("ID", "HMM must have an ID")
46     }
47
48     validationResult.Include(hmm.validateS())
49     validationResult.Include(hmm.validateV())
50     validationResult.Include(hmm.validatePI())
51     validationResult.Include(hmm.validateA())
52     validationResult.Include(hmm.validateB())

```

```
53
54     return validationResult
55 }
56
57 // validateS verifies the validity of the set
58 // of states of a HMM.
59 func (hmm HMM) validateS() ValidationResult {
60     validationResult := ValidationResult{}
61     uniqueElements := map[string]bool{}
62
63     // verify the uniqueness of given states
64     for _, elem := range hmm.S {
65         if uniqueElements[string(elem)] == true {
66             validationResult.Add("S", "States must be unique")
67             break
68         }
69
70         uniqueElements[string(elem)] = true
71     }
72
73     return validationResult
74 }
75
76 // validateV verifies the validity of the set
77 // of observation symbols of a HMM.
78 func (hmm HMM) validateV() ValidationResult {
79     validationResult := ValidationResult{}
80     uniqueElements := map[string]bool{}
81
82     // verify the uniqueness of given observation symbols
83     for _, elem := range hmm.V {
84         if uniqueElements[string(elem)] == true {
85             validationResult.Add("V", "Observation symbols must be unique")
86             break
87         }
88
89         uniqueElements[string(elem)] = true
90     }
91
92     return validationResult
93 }
94
95 // validatePI verifies the validity of the set
96 // of initial states.
97 func (hmm HMM) validatePI() ValidationResult {
98     validationResult := ValidationResult{}
99
100    // verify size (1xN)
101    if len(hmm.PI) != hmm.N {
102        validationResult.Add("PI", "Invalid number of start probabilities;
            must be N or 1 for each state")
103    }
104
105    // verify vector identifier
```

```

106     for key := range hmm.PI {
107         if !isInStateSlice(key, hmm.S) {
108             validationResult.Add("PI", "Given state identifier does not
                exist")
109             break
110         }
111     }
112
113     // verify probabilities
114     for _, value := range hmm.PI {
115         if value < 0 {
116             validationResult.Add("PI", "Probabilities must be > 0")
117             break
118         }
119     }
120
121     // verify that the initial probabilities sum up to 1
122     var probability float64
123     for _, value := range hmm.PI {
124         probability += value
125     }
126
127     if probability != 1.0 {
128         validationResult.Add("PI", "Initial probabilities must sum up to 1.0
                ")
129     }
130
131     return validationResult
132 }
133
134 // validateA verifies the validity of the transition matrix of a HMM.
135 func (hmm HMM) validateA() ValidationResult {
136     validationResult := ValidationResult{}
137
138     // verify matrix size (NxN)
139     if len(hmm.A) == hmm.N {
140         for row := range hmm.A {
141             if cols := hmm.A[row]; len(cols) != hmm.N {
142                 validationResult.Add("A", "Invalid number of columns; must
                    be N")
143                 break
144             }
145         }
146     } else {
147         validationResult.Add("A", "Invalid number of rows; must be N")
148     }
149
150     // verify matrix identifier
151     for row := range hmm.A {
152         if !isInStateSlice(row, hmm.S) {
153             validationResult.Add("A", "Given state identifier of row does
                not exist")
154             break
155         }

```

```

156
157     for col := range hmm.A[row] {
158         if !isInStateSlice(col, hmm.S) {
159             validationResult.Add("A", "Given state identifier of col
160                 does not exist")
161             break
162         }
163     }
164
165     // verify probabilities
166     for row := range hmm.A {
167         for col := range hmm.A[row] {
168             if hmm.A[row][col] < 0 {
169                 validationResult.Add("A", "Probabilities must be > 0")
170                 break
171             }
172         }
173     }
174
175     // verify sum of transition probabilities
176     // All probabilities in each row must sum up to 1
177     // Or each transition from a fixed state si to all other states sj
178     // (assuming that every state can reach every other state) must be 1
179     for row := range hmm.A {
180         var probability float64
181
182         for col := range hmm.A[row] {
183             probability += hmm.A[row][col]
184         }
185
186         if probability != 1.0 {
187             validationResult.Add("A", "All state transition probabilities in
188                 each row must sum up to 1")
189             break
190         }
191     }
192     return validationResult
193 }
194
195 // validateB verifies the validity of the emissions matrix of a HMM.
196 func (hmm HMM) validateB() ValidationResult {
197     validationResult := ValidationResult{}
198
199     // verify matrix size (NxM); number of states x number of observation
200     // symbols
201     if len(hmm.B) == hmm.N {
202         for row := range hmm.B {
203             if cols := hmm.B[row]; len(cols) != hmm.M {
204                 validationResult.Add("B", "Invalid number of columns; must
205                     be M")
206                 break
207             }
208         }
209     }

```

```
206     }
207 } else {
208     validationResult.Add("B", "Invalid number of rows; must be N")
209 }
210
211 // verify matrix identifier
212 for row := range hmm.B {
213     if !isInStateSlice(row, hmm.S) {
214         validationResult.Add("B", "Given state identifier of row does
215             not exist")
216         break
217     }
218     for col := range hmm.B[row] {
219         if !isInObservationSlice(col, hmm.V) {
220             validationResult.Add("B", "Given observation identifier of
221                 col does not exist")
222             break
223         }
224     }
225 }
226
227 // verify probabilities
228 for row := range hmm.B {
229     for col := range hmm.B[row] {
230         if hmm.B[row][col] < 0 {
231             validationResult.Add("B", "Probabilities must be > 0")
232             break
233         }
234     }
235 }
236
237 // verify sum of emission probabilities
238 // All probabilities in each row must sum up to 1
239 // Or for each state all observations must sum up to 1
240 for row := range hmm.B {
241     var probability float64
242     for col := range hmm.B[row] {
243         probability += hmm.B[row][col]
244     }
245
246     if probability != 1.0 {
247         validationResult.Add("B", "All state emission probabilities in
248             each row per state must sum up to 1")
249         break
250     }
251 }
252 return validationResult
253 }
```

Listing 5: hmm.go

Listing 6: sequence.go

```

1 package main
2
3 import "fmt"
4
5 // Sequence represents a succession of observations which will be tested
6 // against Hidden Markov Models in order to determine the likeliest HMM.
7 type Sequence struct {
8     ID          string          `json:"id"`
9     HMMIDs      []string          `json:"hmm_ids"`
10    Observations []ObservationSymbol `json:"observations"`
11 }
12
13 // Validate verifies the validity of the sequence.
14 func (seq Sequence) Validate(hmms []HMM) ValidationResult {
15     validationResult := ValidationResult{}
16
17     if seq.ID == "" {
18         validationResult.Add("ID", "Sequence must have an ID")
19     }
20
21     if len(seq.HMMIDs) == 0 {
22         validationResult.Add("HMMIDs", "No HMM Ids given")
23     }
24
25     // verify that the HMMs which the sequence tests against, exists
26     hmmIDs := hmmIDsToStringSlice(hmms)
27     for _, seqHMMID := range seq.HMMIDs {
28         if !isInStringSlice(seqHMMID, hmmIDs) {
29             validationResult.Add("HMMIDs",
30                 fmt.Sprintf("HMM with ID '%s' does not exist", seqHMMID))
31             break
32         }
33     }
34
35     // verify that the unique observation symbols do exist in all HMMs
36     // the sequence is testing against
37     for _, symbol := range seq.Observations {
38         for _, hmm := range hmms {
39             // test only against the target HMMs
40             if !isInStringSlice(hmm.ID, seq.HMMIDs) {
41                 continue
42             }
43             if !isInObservationSlice(symbol, hmm.V) {
44                 validationResult.Add("Observations",
45                     fmt.Sprintf("Symbol '%q' doesn't exists in target HMMs",
46                         symbol))
47                 break
48             }
49         }
50     }
51     return validationResult
52 }

```

Listing 6: sequence.go

Listing 7: forward.go

```

1 package main
2
3 // Evaluate calculates the likelihood that a HMM generated the observed
  sequence.
4 // It uses the forward-algorithm consisting of three steps - initialization,
5 // induction and termination - to find the likelihood efficiently.
6 func Evaluate(seq Sequence, hmm HMM) float64 {
7     T := len(seq.Observations)
8
9     var probability float64
10    for _, state := range hmm.S {
11        probability += induction(seq, hmm, state, T)
12    }
13
14    return probability
15 }
16
17 // induction calculates for a given state s in a HMM the probability of
18 // reaching the state at point t - by calculating all previous paths to s at
19 // t - and observing the symbol of the sequence at point t
20 func induction(seq Sequence, hmm HMM, s State, t int) float64 {
21     var probability float64
22
23     // tPrev means the previous point while every
24     // array access with t - 1 means at point t with adjusting
25     // to the array indexing order starting from 0
26     tPrev := t - 1
27
28     for _, fromState := range hmm.S {
29         // the probability of the previous point; helper variable
30         var pPrev float64
31
32         // if the previous point is the first, calculate it using PI;
33         // otherwise, do another induction step at point tPrev = t - 1
34         if tPrev > 1 {
35             pPrev = induction(seq, hmm, fromState, tPrev)
36         } else {
37             pPrev = hmm.PI[fromState] *
38                 hmm.B[fromState][seq.Observations[tPrev-1]]
39         }
40
41         // the previous probability at t - 1 given a previous state
42         // multiplied by the probability of actually transition from
43         // the previous state into the actual one at t
44         probability += (pPrev * hmm.A[fromState][s])
45     }
46
47     // all probabilities of reaching state s at point t multiplied by the
48     // probability
49     // of observing the symbol of the sequence at time t given state s
50     return probability * hmm.B[s][seq.Observations[t-1]]
51 }

```

Listing 7: forward.go

Listing 8: structs.go

```

1 package main
2
3 import (
4     "encoding/json"
5 )
6
7 // ValidationResult contains the errors within
8 // a validation process. An error is keyed with an identifier.
9 type ValidationResult map[string]string
10
11 // Add extends the current validation result by another entry
12 func (val ValidationResult) Add(key string, value string) {
13     val[key] = value
14 }
15
16 // Include extends the current validation result by another.
17 func (val ValidationResult) Include(v ValidationResult) {
18     for key, value := range v {
19         val[key] = value
20     }
21 }
22
23 // Valid checks whether the validation result contains no
24 // errors.
25 func (val ValidationResult) Valid() bool {
26     return len(val) == 0
27 }
28
29 // Marshal serialized the validation result to JSON.
30 func (val ValidationResult) Marshal() string {
31     b, err := json.MarshalIndent(val, "", "\t")
32     if err != nil {
33         b = []byte("")
34     }
35
36     return string(b)
37 }
38
39 // Result represents the result of the evaluation of
40 // sequence given individual HMMs. The ID is the ID of the HMM.
41 type Result struct {
42     ID            string
43     Likelihood     float64
44     LogLikelihood float64
45 }
46
47 // Better determines by comparing it with another result whether
48 // the current result is better.
49 func (res Result) Better(x Result) bool {
50     return res.Likelihood > x.Likelihood
51 }

```

Listing 8: structs.go



Listing 9: utils.go

```

1 package main
2
3 // isInStateSlice checks whether a given state is
4 // in an array of states.
5 func isInStateSlice(s State, slice []State) bool {
6     for _, elem := range slice {
7         if elem == s {
8             return true
9         }
10    }
11
12    return false
13 }
14
15 // isInStateSlice checks whether a given observation symbol is in
16 // an array of observation symbols.
17 func isInObservationSlice(s ObservationSymbol, slice []ObservationSymbol)
18     bool {
19     for _, elem := range slice {
20         if elem == s {
21             return true
22         }
23     }
24
25     return false
26 }
27
28 // isInStringSlice checks whether a given string x is in an
29 // array of strings.
30 func isInStringSlice(x string, elements []string) bool {
31     for _, elem := range elements {
32         if elem == x {
33             return true
34         }
35     }
36
37     return false
38 }
39
40 // hmmIDsToStringSlice extracts the ID of each given HMM
41 // and returns it as an array.
42 func hmmIDsToStringSlice(hmms []HMM) []string {
43     ids := []string{}
44     for _, hmm := range hmms {
45         ids = append(ids, hmm.ID)
46     }
47
48     return ids
49 }

```

Listing 9: utils.go