Artificial Intelligence - HMM Assignment

Oliver Midbrink, Christoph Siebürger

```
def matmul(A, B):
    if len(A[0]) != len(B): # A col number is not B row number
        return None

    output_size = (len(A), len(B[0]))

    output_mat = []
    for row in range(output_size[0]):
        row_ = []
        for col in range(output_size[1]):
            element_value = 0
            for summation_idx in range(len(B)):
                element_value += A[row][summation_idx] * B[summation_idx][col]
            row_.append(element_value)
        output_mat.append(row_)

    return output_mat

def transpose(A):
    output_mat = []

    for col in range(len(A[0])):
        row_ = []
        for row in range(len(A)):
            row_.append(A[row][col])
        output_mat.append(row_)

    return output_mat

def to_log(A):
    A_log = copy.deepcopy(A)
    for row in range(len(A)):
        for col in range(len(A[0])):
            if A_log[row][col] == 0:
                A_log[row][col] = float("-inf")
            else:
                A_log[row][col] = math.log(A_log[row][col])
    return A_log
```

```
def from_log(A):
    A_norm = copy.deepcopy(A)
    for row in range(len(A)):
        for col in range(len(A[0])):
            A_norm[row][col] = math.exp(A[row][col])
    return A_norm

def _sum(A):
    sum_ = 0
    for row in range(len(A)):
        for col in range(len(A[0])):
            sum_ += A[row][col]
    return sum_

def one_line_matrix(A):
    output_line_comps = [len(A), len(A[0])]

    for row in range(len(A)):
        for col in range(len(A[0])):
            output_line_comps.append(A[row][col])
```

Above you see some functions developed primarily for hmm0 (because transpose and matmul was not used in later labs.(the other ones, like from_log and to_log, were used readily). one_line_matrix was used a lot for hmm0-3 to print out the matrix results. All functions were designed with 2d matrices in mind. even column and row vectors would be treated as 2d matrices, to ensure smooth interfacing between functions and reduce if else statements needed.

```
def alpha_pass(trans, emission, initial, o_seq):
    n_x = len(trans)
    n_o_ = len(o_seq)

    epsilon = float("-inf")

    alpha_vector = [[epsilon] * n_x for _ in range(n_o_)]

    # initialise alpha_vector
    for x in range(n_x):
        alpha_vector[0][x] = initial[0][x] + emission[x][o_seq[0]]

    for t_ in range(1, n_o_, 1):
        for x_j in range(n_x):
            for x_i in range(n_x):
```

```
                alpha_vector[t_][x_j] = log_sum_exp(alpha_vector[t_][x_j],
alpha_vector[t_-1][x_i] + trans[x_i][x_j])
            alpha_vector[t_][x_j] += emission[x_j][o_seq[t_]]

    return alpha_vector

def beta_pass(trans, emission, initial, o_seq):
    n_x = len(trans)
    n_o_ = len(o_seq)

    epsilon = float("-inf")
    beta_vector = [[epsilon] * n_x for _ in range(n_o_)]

    # initialise beta_vector
    for x in range(n_x):
        beta_vector[-1][x] = 0

    for t_ in range(n_o_-2, -1, -1):
        for x_i in range(n_x):
            for x_j in range(n_x):
                beta_vector[t_][x_i] = log_sum_exp(beta_vector[t_][x_i],
beta_vector[t_+1][x_j] + trans[x_i][x_j] + emission[x_j][o_seq[t_+1]])

    return beta_vector
```

Above are some of the HMM relevant algorithms. For example shown is the alpha and beta pass. Since we are working with log probabilities, and we want to sum the probability from different paths, we need to find a way to sum log probabilities. This was not trivial, because you need to ensure numerical stability, while treating edge cases like float(“inf”). Here below is the function for the log_sum_exp function (it essentially does log(exp(a) + exp(b)) to be able to sum log probabilities like you would with normal probabilities, however it employes the extra steps mentioned. To better find a solution, we looked up online (wikipedia) how to add log probabilities in a numerically stable way. Furthermore, the choice of using the log space formulations of alpha and beta pass (as well as gamma, di-gamma etc) are to reduce underflow, since the last alpha vector row can be -2000 in some cases (which would not even have a non-zero value in normal floats.

```
def log_sum_exp(a, b):
    if a == float('-inf'):
        return b
    if b == float('-inf'):
        return a
    if a > b:
        return a + math.log1p(math.exp(b - a))
    else:
```

```
        return b + math.log1p(math.exp(a - b))
```

Below is our di-gamma function:

```
def di_gamma(t, i, j, a, b, alpha, beta, obs_seq, alpha_lower_sum):
    upper = alpha[t][i] + a[i][j] + b[j][obs_seq[t+1]] + beta[t+1][j]
    return upper - alpha_lower_sum
```

Instead of recalculating the alpha_lower_sum for every time we call it (many times) we calculate the alpha_lower_sum in the main loop and pass its pointer through our arguments in a nested fashion, main loop → get_di_gamma_mat → di_gamma(t, i, j)

Similarly the get a and get b functions use a similar sum (but in this case a vector, values for each state) to get the a and b matrix. We calculate the sum of gamma over the timesteps for each state in the main loop, call it gamma_cat_sum and pass it from the main loop into the arguments of the functions that calculate the a and b estimates. This enables fast computation and was crucial to be able to pass hmm3 assignment, without it the program was way too slow.

```
def get_abc(di_gamma_mat, gamma_mat, gamma_cat_sum, obs_seq, a, b, c):
    a_new = copy.deepcopy(a)
    b_new = copy.deepcopy(b)
    c_new = copy.deepcopy(c)

    for row in range(len(a)):
        for col in range(len(a[0])):
            a_new[row][col] = _a(row, col, di_gamma_mat, gamma_mat, gamma_cat_sum,
obs_seq)

        for col in range(len(b[0])):
            b_new[row][col] = _b(row, col, gamma_mat, gamma_cat_sum, obs_seq)

        c_new[0][row] = _c(row, gamma_mat)

    return a_new, b_new, c_new
```

Get abc above is a wrapper function that gets the next estimate for a b and c. copy.deepcopy(mat) will copy all of the elements at multiple depths into a new memory space, reducing the risk of changing the original matrix (otherwise dim 1 would reference the other matrix). (removing this will yield 1 less score in kattis)

```
def p_obs_given_abc(o_seq, a, b, c):
    alpha = alpha_pass(a, b, c, o_seq)

    alpha_lower_sum = float("-inf")
    for x in alpha[-1]:
```

```
        alpha_lower_sum = log_sum_exp(alpha_lower_sum, x)
    return alpha_lower_sum
```

Here above we have a functions to calculate the probability of observing a sequence given a model (lambda = (a, b, c)) This is useful in the derby assignment:

```
def fit_HMM(o_seq, n_obs, n_states=None, a=None, b=None, c=None):
    n_o = len(o_seq)
    if n_states is not None:
        a = to_log(uniform_mat(n_states, n_states))
        b = to_log(uniform_mat(n_states, n_obs))
        c = to_log(uniform_mat(1, n_states))

    lr = 1

    count = 0
    alpha_lower_sum = float("-inf")
    converged = False
    while True:

        alpha = alpha_pass(a, b, c, o_seq)
        beta = beta_pass(a, b, c, o_seq)

        alpha_lower_sum = float("-inf")
        for x in alpha[-1]:
            alpha_lower_sum = log_sum_exp(alpha_lower_sum, x)

        di_gamma_mat = get_di_gamma(a, b, alpha, beta, o_seq, alpha_lower_sum)
        gamma_mat = get_gamma(di_gamma_mat, alpha, o_seq)

        gamma_cat_sum = [float("-inf")] * n_o
        for x in range(len(a)):
            for t_ in range(n_o - 1):
                gamma_cat_sum[x] = log_sum_exp(gamma_cat_sum[x], gamma_mat[t_][x])

        a_expect, b_expect, c_expect = get_abc(di_gamma_mat, gamma_mat,
gamma_cat_sum, o_seq, a, b, c)

        def update_mat(est_mat, old_mat, lr_, conv_log=True):
            if conv_log:
                return to_log(element_wise_add(scalar_mat_mul(lr_,
from_log(est_mat)), scalar_mat_mul((1 - lr), from_log(old_mat))))
            else:
                return element_wise_add(scalar_mat_mul(lr_, est_mat),
scalar_mat_mul((1 - lr), old_mat))
```

```
        a_new, b_new, c_new = update_mat(a_expect, a, lr), update_mat(b_expect, b,
lr), update_mat(c_expect, c, lr)

        #print("Change", change)

        diff = eab_d(a, a_expect) + eab_d(b, b_expect) + eab_d(c, c_expect)
        a, b, c = a_new, b_new, c_new

        #if math.isnan(diff):
        #    print(a, b, c)
        #    print(o_seq)

        count += 1
        #print("times: ", t_1, t_2, t_3, t_4, t_5, t_6)
        if count == 120:
            break
        #print(diff)

        if diff < 1e-4:
            converged=True
            break

    return a, b, c, converged
```

Above is another useful function which will fit, or fine tune an HMM model. In the end we chose never to use the fine tune mode, because its better to add sequences together into one, start with uniform distribution (parameters) and then converge in just on or two steps (very fast convergence due to uniform initial dist), instead of tuning the parameters by starting at the convergence of one sequence when learning the next (slow convergence).

The functions will also return whether something happened during convergence, for example “nan”. Which we can then use in a loop (where we regularise the sequence, for example with laplace label adding and random label adding) in order to give the model another chance to converge. This is because we want all the models to have converged so we can best guess the most likely fish.

```
class PlayerControllerHMM(PlayerControllerHMMAbstract):
    def init_parameters(self):
        """
        In this function you should initialize the parameters you will need,
        such as the initialization of models, or fishes, among others.
        """
        self.fish_obs_seqs = [[] for _ in range(N_FISH)]
        self.fish_types_result_id_dict = {}
        self.models = {}
```

```
        self.step = 0
        self.states = 3

    def get_unvalidated_fish(self):
        unvalidate_fish_ids = []
        for fish_id in range(N_FISH):
            if fish_id not in self.fish_types_result_id_dict:
                unvalidate_fish_ids.append(fish_id)
        return unvalidate_fish_ids

    def get_best_type(self, fish_id):
        max_prob = float("-inf")
        type_best = -1

        for fish_type in range(N_SPECIES):
            if fish_type in self.models.keys():
                a, b, c = self.models[fish_type]
                prob = p_obs_given_abc(self.fish_obs_seqs[fish_id], a, b,
c)/self.step
            else:
                prob = float("-inf")

            if prob >= max_prob:
                max_prob = prob
                type_best = fish_type

        return type_best, max_prob

    def fish_found_ever(self):
        return len(self.fish_types_result_id_dict) > 0

    def find_best_next_fish(self):
        total_best_prob = float("-inf")
        total_best_guess = -1

        for fish_id in self.get_unvalidated_fish():
            best_guess, best_prob = self.get_best_type(fish_id)

            if best_prob >= total_best_prob:
                total_best_prob = best_prob
                total_best_guess = (fish_id, best_guess)

        return total_best_guess, total_best_prob

    def guess(self, step, observations):
```

```
        """
        This method gets called on every iteration, providing observations.
        Here the player should process and store this information,
        and optionally make a guess by returning a tuple containing the fish index
and the guess.
        :param step: iteration number
        :param observations: a list of N_FISH observations, encoded as integers
        :return: None or a tuple (fish_id, fish_type)
        """
        self.step = step
        # This code would make a random guess on each step:
        for fish_idx, obs in enumerate(observations):
            self.fish_obs_seqs[fish_idx].append(obs)

        if self.fish_found_ever():
            best_guess, prob = self.find_best_next_fish()
            return best_guess


        if step == N_STEPS - N_FISH - 0:
            return (0, 0)

        return None

    def add_missing_species(self, sequence, alpha=30, noise=25):
        missing = [i for i in range(N_SPECIES) if i not in sequence]
        added = []
        for species in missing:
            for _ in range(alpha):
                added.append(species)
        noise_added = random.choices(range(N_SPECIES), k=noise)
        return sequence + added + noise_added


    def reveal(self, correct, fish_id, true_type):
        """
        This methods gets called whenever a guess was made.
        It informs the player about the guess result
        and reveals the correct type of that fish.
        :param correct: tells if the guess was correct
        :param fish_id: fish's index
        :param true_type: the correct type of the fish
        :return:
        """
        self.fish_types_result_id_dict[fish_id] = true_type
```

```
        if true_type not in self.models:
            a, b, c, conv =
fit_HMM(self.add_missing_species(self.fish_obs_seqs[fish_id]), N_EMISSIONS,
self.states)

            count = 0
            while not conv:
                len_ = len(self.fish_obs_seqs[fish_id])
                seq_start = random.randint(0, len_ - int(len_ * 0.97) - 1)
                seq = self.fish_obs_seqs[fish_id][seq_start:seq_start+len_]
                a, b, c, conv = fit_HMM(seq, N_EMISSIONS, self.states)
                count += 1

                if count > 2:
                    a, b, c, conv =
fit_HMM(random.sample(self.add_missing_species(seq),
len(self.add_missing_species(seq))-1), N_EMISSIONS, self.states)

            self.models[true_type] = (a, b, c)
        else:
            total_obs = []
            for fish_id_, fish_type in self.fish_types_result_id_dict.items():
                if fish_type == true_type:
                    total_obs += self.fish_obs_seqs[fish_id_]

            a, b, c, conv = fit_HMM(self.add_missing_species(total_obs),
N_EMISSIONS, self.states)
            while not conv:
                len_ = len(total_obs)
                seq_start = random.randint(0, len_ - int(len_ * 0.99) - 1)
                seq = total_obs[fish_id][seq_start:seq_start+len_]
                a, b, c, conv = fit_HMM(random.sample(self.add_missing_species(seq),
len(self.add_missing_species(seq))-1), N_EMISSIONS, self.states)

            self.models[true_type] = (a, b, c)
```

This above is our code for the HMM derby. As you can see we do the following, wait as long as possible and record all observations at every timestep (gather data).

At timestep N_STEPS - N_FISH, start guessing so we make a guess for every fish. When making a guess, start with random (since we do not know, and need samples to identify). We could do a correlation matrix of the crossover fish probs using HMMs for each fish, choose the biggest cluster and guess this one, to get the most data at the first time. But our simple approach was enough to pass with a margin.

So basically when we know one of the fish, we train an HMM model on its movement. After that we look at all the fish to see using the p_obs_given_abc function to see which is most similar to it, that fish we guess has the same type. For fish that is added, if its the same type as we have seen before, we merge their observation sequence for more training data. Otherwise we create a new model.

When we have multiple models, we see which model on which fish (2d for loop) has the highest probability. This is our next guess. Equivalently, we can guess on the worst fish every time, in the end we get the same score. (as long as we see which model is closest to that fish). This is because the model will always use all available and labeled data when training an HMM and using it for prediction, and the underlying data will be the previous guesses (which provide labels)

The add missing species function gives us a few extra points by adding random observations (for convergence stability and generalisability, as well as laplace like adding of missing species). Here the random extra labels yielded the biggest part of the improvement.

We tried using different numbers of states, which yielded exactly the same result. This led us to the hypothesis that the states do not matter. Hence, one could perhaps approximate the emission matrix with one state by observing the observation sequence proportions and use the euclidian distance between such (k length = N_EMISSIONS) vectors to find similar fish as ones previously discovered. This led us to test the simplified approach (no HMM at all) which yielded the same score as us, but with a much shorter runtime of 1 second instead of 1.4 with a maximally efficient HMM implementation. However one piece was still missing, we were making the best possible guesses, but our results were still not perfect. The only way i know how to solve this would be to see which type of fish are most similar using data science. then repeatedly submitting solutions with parameters such as (move the ith uncertain fish to its second closest subgroup etc.

Basically, we parameter estimation on the known (revealed fish), and used evaluation to find likely similar fish. We chose to implement matrix operations, and logic using standard python lists, because it would be faster according to the assignment description.

Personal reflection Oliver: The project was hard, but you have to divide up the components and look at each one from different angles to make sure it absolutely works. Otherwise if you have a big system that does not work, you have to guess where it does not work. If you are certain about each component, it's much easier to find errors. Also because if you inspect them you understand them better. For example, one unexpected error was that we expected some matrices to be copied before being modified. but if you just pass the matrix and do .copy() it will be a shallow copy (only along the first dimension) however the lists inside the bigger lists, would be just pointers to the original list. This made the program behavior weird, using copy.deepcopy(Matrix) worked like a charm. We try to use this minimally because it uses some time to copy the matrix. We discovered that using more or fewer states in the fishing derby would not yield different results, therefore we realised the game is not dependent on states. We tried using no states (one state), and just observation probabilities as a framework, and it worked just as well but faster. This was a side experiment. Our main code still followed the HMM baum welch framework with 3 states (Tired, active, neutral).

Personal reflection Christoph: The program was slow, so we had to calculate a gamma(i) vector and an alpha last timestep sum (probability of sequence), and store these when calculating the di gamma vector (alpha) and the new A and B matrix. This made the program around twice as fast because these were bottleneck parts. To test the bottlenecks, we recorded timestamps for each function in the loop, and could see where the time increased the most. This was ABC parameter calculation and di gamma matrix calculation, because a lot of the same operations were repeated, which we were able to speed up. Furthermore, we observed that using uniform initialisation matrices would yield convergence in just two steps to a very high degree of convergence, compared to very slow convergence for non uniform parameters. Therefore when we estimate parameters when a new fish is revealed, we instead add all the sequences together into one long, and use uniform convergence on the longer one, instead of reusing old parameters. This was much faster, with similar performance.