For storing initial probabilities, transition and emission matrixes I chose numpy arrays. They are very time efficient, but they need to have size of the arrays specified when we create arrays and elements can only be accessed using integer indexes.

To solve these two problems I have first loop over all training files. Firstly, I create sets of words and tags to get the shape of future numpy arrays. Secondly, I create dictionaries that map:

word -> word_numerical_index,

tag -> tag_numerical_index.

```
tags_set = set()
words_set = set()
tags_dict = dict()
words_dict = dict()
```

In loop I add items to set first. If the size of set changes, I add items to dictionary too.

```
words_set.add(line_l[0])
if len(words_set) != words_len:
    words_dict[line_l[0]] = words_len
    words_len += 1
tags_set.add(line_l[2])
if len(tags_set) != tags_len:
    tags_dict[line_l[2]] = tags_len
    tags_len += 1
```

Then I initialize arrays I needed and go through training files one more time adding values to arrays.

```
initial_vector = np.zeros(len(tags_set), dtype='uint32')
transition_matrix = np.zeros((len(tags_set), len(tags_set)), dtype='uint32')
emission_matrix = np.zeros((len(tags_set), len(words_set)), dtype='uint32')
dot_indx = words_dict['.']
for file_name in training_list:
    with open(file_name, encoding="utf8") as f:
        starting = True
        line = f.readline()
        while line != '':
            word_str, _, tag_str = line.split()
            tag = tags_dict[tag_str]
            word = words_dict[word_str]
            if starting:
                initial_vector[tag] += 1
            else:
                transition_matrix[prev, tag] += 1
            emission_matrix[tag, word] += 1
            prev = tag
            starting = True if word == dot_indx else False
            line = f.readline()
```

Next, I make probabilities from numbers of occurrences by dividing every element on sum of possible variants. In transition, for instance, I take the sum all occurrences of tags after every tag. Then I return matrixes and dictionaries, which would be useful for testing.

```
s = np.sum(initial_vector)
initial_vector = np.divide(initial_vector, s)
transition_matrix = transition_matrix.transpose()
sum_vector = np.sum(transition_matrix, axis=0)
transition_matrix = np.divide(transition_matrix, sum_vector).transpose()
emission_matrix = emission_matrix.transpose()
sum_vector = np.sum(emission_matrix, axis=0)
emission_matrix = np.divide(emission_matrix, sum_vector).transpose()
return (initial_vector, transition_matrix, emission_matrix, words_dict, tags_dict)
```

When making Viterbi function I just followed the pseudocode that I saw on lectures with adding some small features.

I added a check whether the word is in the word dictionary. If not I set it's probability to 1 for every tag, so that it don't affects the result.

```
sentence = [words_dict[i] if i in words_dict else -1 for i in sentence]
```

I also split the calculating of prev and prob matrix. Firstly, for every tag I'm searching the best previous tag. Here I don't take into account the emission matrix probabilities because they are same for every tag.

```
for_prev = np.array([np.argmax(prob[n-1] * transition[:, i]) for i in range(tags_n)])
# dont need emission matrix here because it's constant
```

Secondly, I just calculate probabilities for every tag with using the information about best previous tag and normalize them.

```
for_prob = np.array([
    prob[n-1, for_prev[i]] * transition[for_prev[i], i] * word_probabilities[i] for i in range(tags_n)
    ])
for_prob /= np.sum(for_prob)
prev[n] = for_prev
prob[n] = for_prob
```

Finally, I extract the best path for sentence.

```
path = []
while current_tag >= 0:
    path.append(current_tag)
    current_tag = int(prev[indx, current_tag])
    indx -= 1
path.reverse()
return path
```

Testing function just translates path in tags indexes and writes it to output file.

```
def test(test_file, output_file, train_output):
    initial_vector, transition_matrix, emission_matrix, words_dict, tags_dict = train_output
    tags_l = [0 for _ in range(len(tags_dict))]
    for k,v in tags_dict.items():
        tags_l[v] = k
    with open(test_file, 'r', encoding="utf8") as f_t, open(output_file, 'w', encoding="utf8") as f_o:
        sentence = []
        line = f_t.readline()[:-1]
        while line != '':
            sentence.append(line)
            if line == '.':
                path = viterbi(sentence, initial_vector, transition_matrix, emission_matrix, words_dict)
                lines = [ f"{sentence[i]} : {tags_l[path[i]]}\n" for i in range(len(sentence))]
                f_o.writelines(lines)
                sentence = []
            line = f_t.readline()[:-1]
```