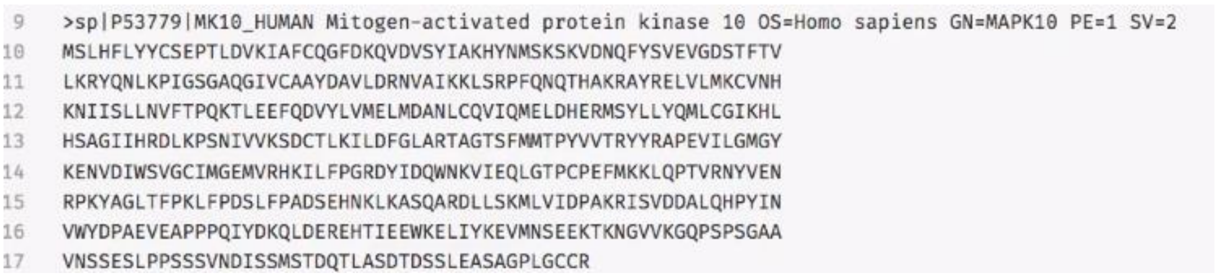
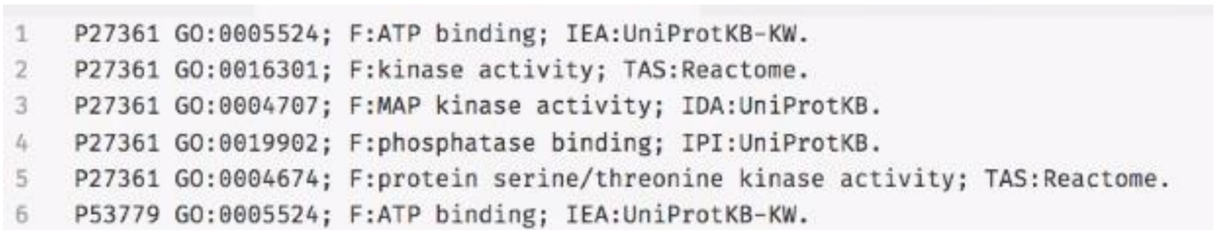
Problem Description and Data View

We wanted to predict if the protein performs the ATP binding functions by their amino acid sequence. Training set consists of amino acid sequences, test set consists of “GO” column.



```
>sp|P53779|MK10_HUMAN Mitogen-activated protein kinase 10 OS=Homo sapiens GN=MAPK10 PE=1 SV=2
MSLHFLYYCSEPTLDVKIAFCQGFDKQVDVSYIAKHYNMSKSKVDNQFYSVEVGDSTFTV
LKRYQNLKPIGSGAQGIVCAAYDAVLDRNVAIKKLSRPFQNQTHAKRAYRELVLMKCVNH
KNIISLLNVFTPQKTLEEFQDVYLVMELMDANLCQVIQMELDHERMSYLLYQMLCGIKHL
HSAGIIHRDLKPSNIVVKSDCTLKILDFGLARTAGTSFMMTPYVVTRYYRAPEVILGMGY
KENVDIWSVGCIMGEMVRHKILFPGRDYIDQWNKVIEQLGTPCPEFMKKLQPTVRNYVEN
RPKYAGLTFPKLFPDSLFPADSEHNKLKASQARDLLSKMLVIDPAKRISVDDALQHPYIN
VWYDPAEVEAPPPQIYDKQLDEREHTIEEWKELIYKEVMNSEEKTKNGVVKGQPSPSGAA
VNSSESLPPSSSVNDISSMSTDQTLASDTDSSLEASAGPLGCCR
```

“p53779” is a protein id. Protein is composed of amino acids. Last row are amino acids that protein is composed of.



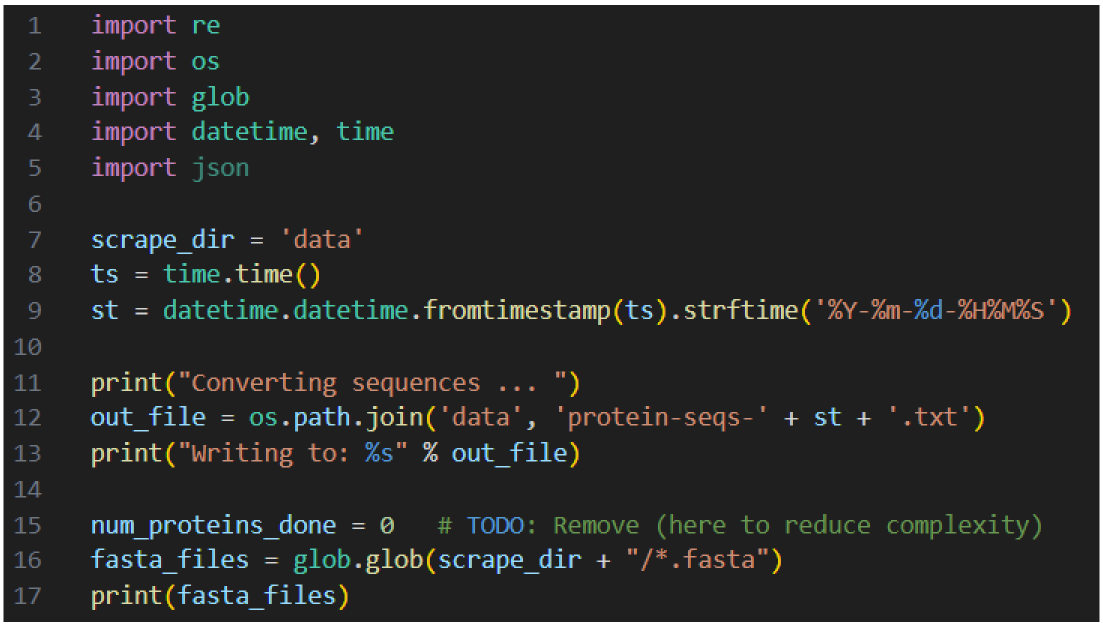
```
P27361 GO:0005524; F:ATP binding; IEA:UniProtKB-KW.
P27361 GO:0016301; F:kinase activity; TAS:Reactome.
P27361 GO:0004707; F:MAP kinase activity; IDA:UniProtKB.
P27361 GO:0019902; F:phosphatase binding; IPI:UniProtKB.
P27361 GO:0004674; F:protein serine/threonine kinase activity; TAS:Reactome.
P53779 GO:0005524; F:ATP binding; IEA:UniProtKB-KW.
```

In another data file, protein “p27361” does the functions: ATP binding, kinase activity, etc. “GO” is the code of the function.

Other proteins can perform the same functions too.

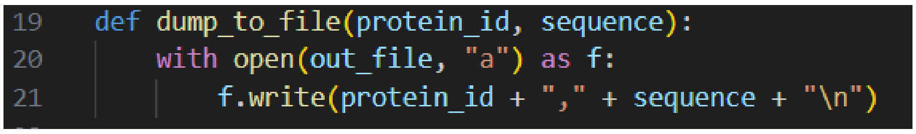
Before applying machine learning models, we need to preprocess the data. Columns that are related with machine learning model should be taken. One protein’s amino acid is written on more than one line. We can’t feed amino acid data to the model this way. We need to take protein id and amino acid sequence from first file. We need to take protein Ids of protein that do the ATP binding. Our dependant value is

Preparing The Data Files



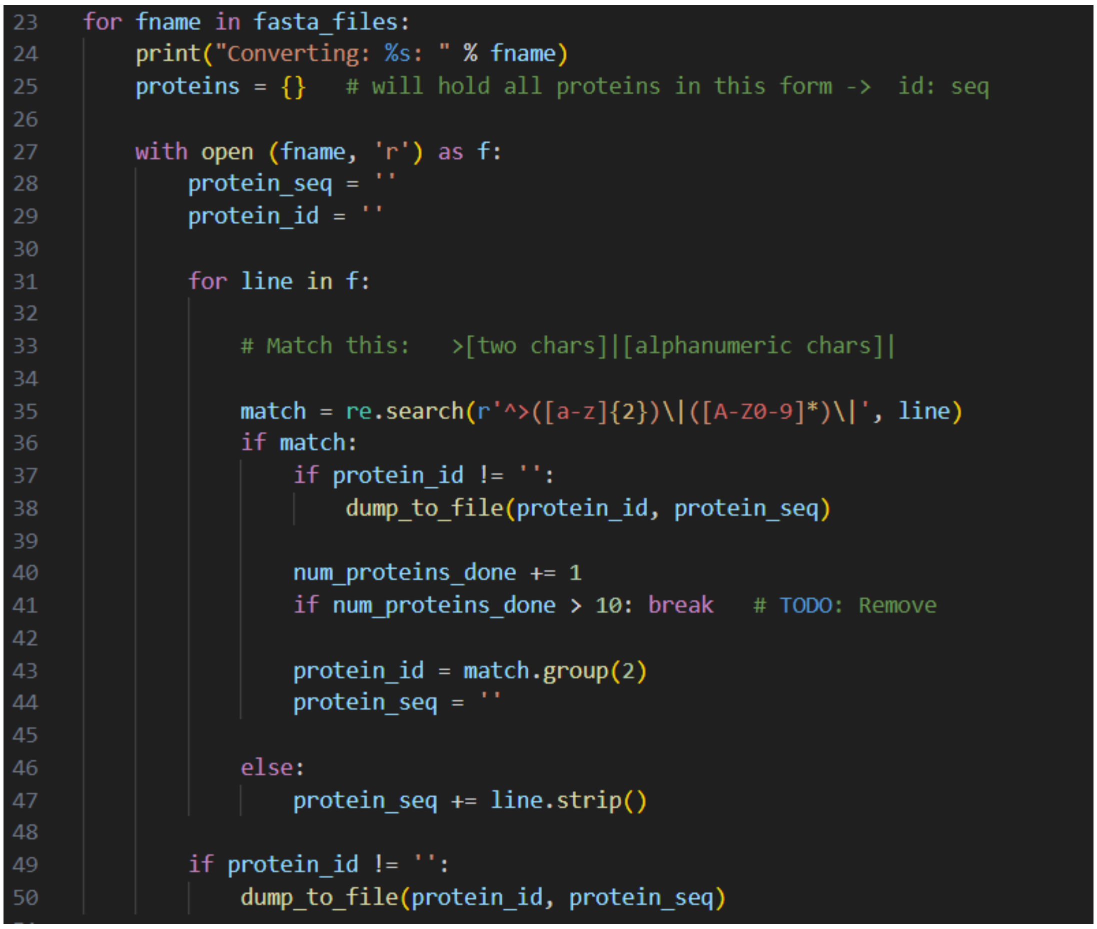
```python
1   import re
2   import os
3   import glob
4   import datetime, time
5   import json
6
7   scrape_dir = 'data'
8   ts = time.time()
9   st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d-%H%M%S')
10
11  print("Converting sequences ... ")
12  out_file = os.path.join('data', 'protein-seqs-' + st + '.txt')
13  print("Writing to: %s" % out_file)
14
15  num_proteins_done = 0   # TODO: Remove (here to reduce complexity)
16  fasta_files = glob.glob(scrape_dir + "/*.fasta")
17  print(fasta_files)
```

- Line 8: Takes current time.
- Line 9: Saves the taken time in a specific format.
- Line 12: Creates output file’s path.
- Line 15: Limiting lines for trying the code if it works or not.
- Line 16: Takes all of the fasta files in the path. “/*.fasta” means it is going to take all of the fasta files.



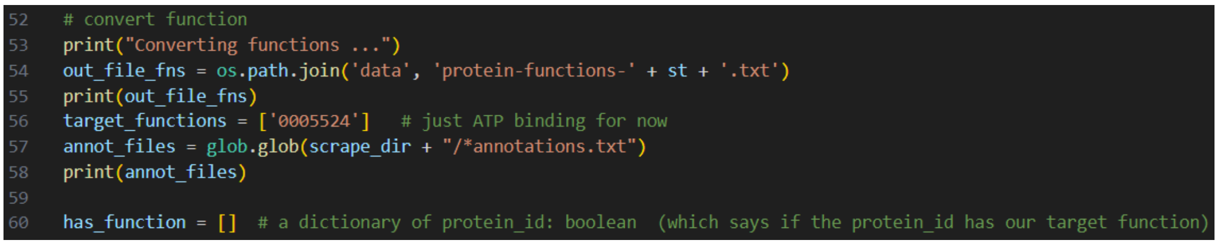
```python
19  def dump_to_file(protein_id, sequence):
20      with open(out_file, "a") as f:
21          f.write(protein_id + "," + sequence + "\n")
```

- Line 19-21: Function that writes protein id and protein sequence to output file.



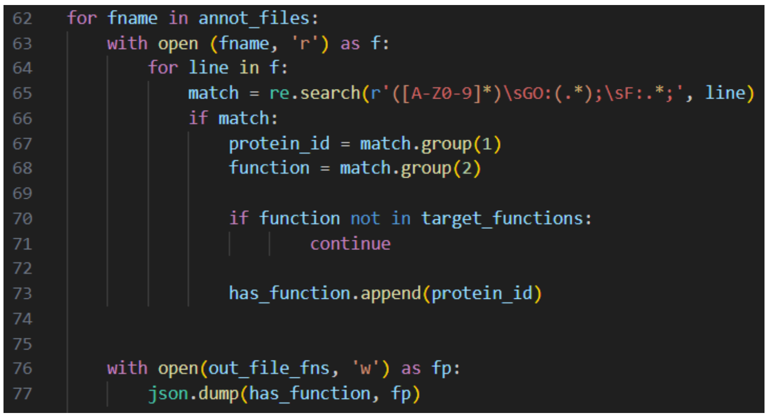
```
23  for fname in fasta_files:
24      print("Converting: %s: " % fname)
25      proteins = {}   # will hold all proteins in this form ->  id: seq
26
27      with open (fname, 'r') as f:
28          protein_seq = ''
29          protein_id = ''
30
31          for line in f:
32
33              # Match this:   >[two chars]|[alphanumeric chars]|
34
35              match = re.search(r'^>([a-z]{2})\|([A-Z0-9]*)\|', line)
36              if match:
37                  if protein_id != '':
38                      dump_to_file(protein_id, protein_seq)
39
40                  num_proteins_done += 1
41                  if num_proteins_done > 10: break    # TODO: Remove
42
43                  protein_id = match.group(2)
44                  protein_seq = ''
45
46              else:
47                  protein_seq += line.strip()
48
49          if protein_id != '':
50              dump_to_file(protein_id, protein_seq)
```

- Line 23: Looping through every fasta files.
- Line 27: Opens the current fasta file for reading purposes.
- Line 28: Variable that stores amino acid sequences.
- Line 29: Variable that stores id of protein.
- Line 31: Loops through every line of the fasta file.
- Line 35: This variable that contains every row's format. Lets take the first line's values: ">sp|P53779". "^[a-z]{2}" means that there are two alphabeticall character in the first group starts with '^' like "^sp". "\" is an escape character, so in order to write "|" we apply "\|". "[A-Z0-9]*" means that there are unknown length of string consists of alphabetical and numerical characters exists as group 2 like "P53779".
- Line 36: If format matches start to process.
- Line 37: If "protein_id" variable isn't empty, it means that it already got some value in the previous step of the loop. We call the "dump_to_file" function to save this variables to the output file.
- Line 40-41: Controls if the number of proteins exceeds the limit or not.
- Line 43: Gets the protein id from match variables group 2. Which is [A-Z0-9]*.
- Line 46: If it doesn't match the format, it means that the line contains amino acid sequence. Takes every line of these sequence and saves it to "protein_seq".
- Line 50: Saves the last remaining variable to the output file.



```python
52  # convert function
53  print("Converting functions ...")
54  out_file_fns = os.path.join('data', 'protein-functions-' + st + '.txt')
55  print(out_file_fns)
56  target_functions = ['0005524']   # just ATP binding for now
57  annot_files = glob.glob(scrape_dir + "/*annotations.txt")
58  print(annot_files)
59
60  has_function = []  # a dictionary of protein_id: boolean  (which says if the protein_id has our target function)
```

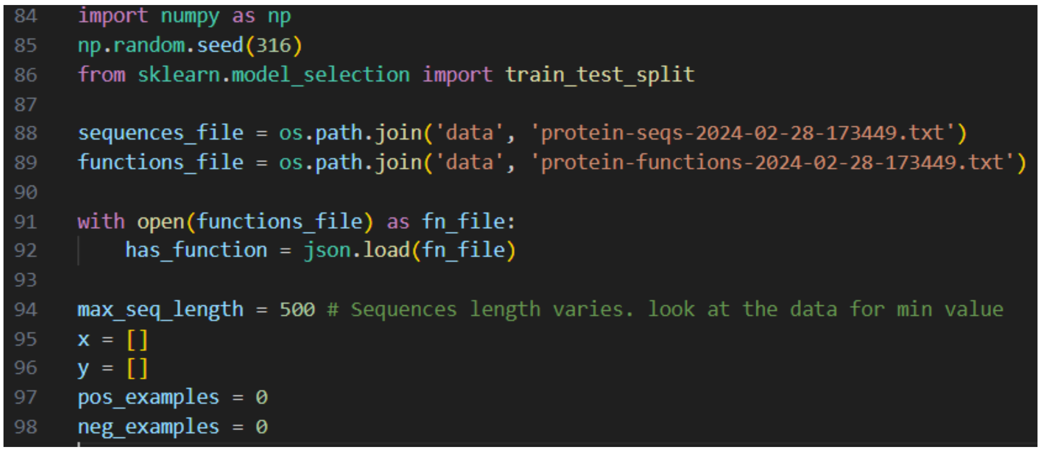
- Line 54: Variable that stores the path of our second output file.
- Line 56: List that stores wanted function’s code. In out case it is ATP binding.
- Line 58: Variable that stores imported file’s data. This list holds every files’ value which ends with “annotations.txt”.
- Line 60: If a protein can perform target function, ID of this protein is going to be saved here.



```python
62  for fname in annot_files:
63      with open (fname, 'r') as f:
64          for line in f:
65              match = re.search(r'([A-Z0-9]*)\sGO:(.*);\sF:.*;', line)
66              if match:
67                  protein_id = match.group(1)
68                  function = match.group(2)
69
70                  if function not in target_functions:
71                          continue
72
73                  has_function.append(protein_id)
74
75
76      with open(out_file_fns, 'w') as fp:
77          json.dump(has_function, fp)
```

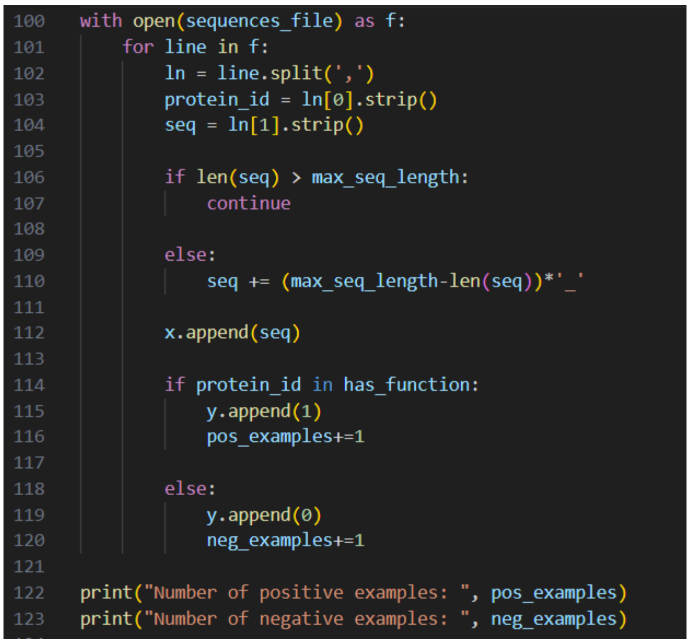
- Line 62-63: Reads every file in the list.
- Line 64: Loops through every line of the file.
- Line 65: Defining a variable to control if format is matching.
- Line 66: Controlling if the format is matching.
- Line 67: Taking the group 1’s value as protein ID.
- Line 68: Taking the group 2’s value as function.
- Line 70-73: If the function is in the target function list, write this protein’s ID into list.
- Line 76-77: Writing the values of this list into json file.

Preprocessing



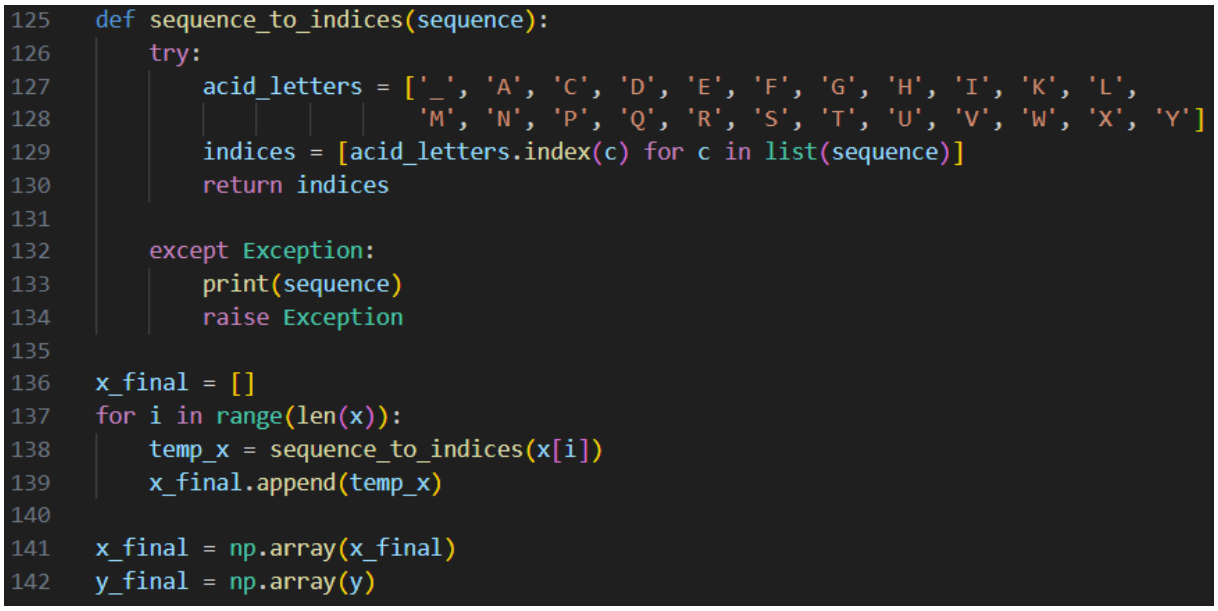
```python
84  import numpy as np
85  np.random.seed(316)
86  from sklearn.model_selection import train_test_split
87
88  sequences_file = os.path.join('data', 'protein-seqs-2024-02-28-173449.txt')
89  functions_file = os.path.join('data', 'protein-functions-2024-02-28-173449.txt')
90
91  with open(functions_file) as fn_file:
92      has_function = json.load(fn_file)
93
94  max_seq_length = 500 # Sequences length varies. look at the data for min value
95  x = []
96  y = []
97  pos_examples = 0
98  neg_examples = 0
```

- Line 88: Path of the protein sequence file into variable.
- Line 89: Path of the protein functions file into variable.
- Line 91-92: Loading the protein functions file into variable.
- Line 94: Length of the each sequence varies. With this value, length is fixed to 500.



```python
100 with open(sequences_file) as f:
101     for line in f:
102         ln = line.split(',')
103         protein_id = ln[0].strip()
104         seq = ln[1].strip()
105
106         if len(seq) > max_seq_length:
107             continue
108
109         else:
110             seq += (max_seq_length-len(seq))*'_'
111
112         x.append(seq)
113
114         if protein_id in has_function:
115             y.append(1)
116             pos_examples+=1
117
118         else:
119             y.append(0)
120             neg_examples+=1
121
122 print("Number of positive examples: ", pos_examples)
123 print("Number of negative examples: ", neg_examples)
```

- Line 100-101: Opens the sequence file and loops through every line.
- Line 102: Split the line with ‘,’ operator.
- Line 103: First element is protein ID.
- Line 104: Second element is amino acid sequence.
- Line 106-107: Continues if the length of the sequence is more than wanted length.
- Line 109-110: Length of every sequence varies. If the length of sequence is less than wanted, ‘_’ is added to sequence until the wanted length is reached.
- Line 112: Sequence is added to the x list.
- Line 114-120: If protein is in listed functions, 1 value is added into the y list. Else 0 value is added into y list.



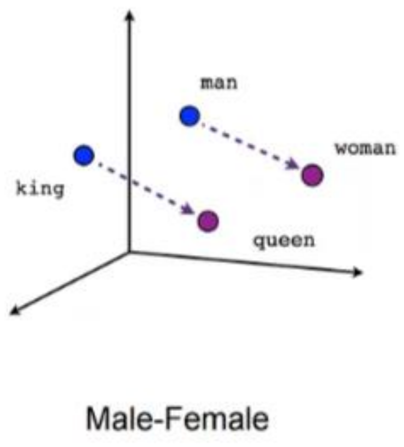
```
125  def sequence_to_indices(sequence):
126      try:
127          acid_letters = ['_', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L',
128                          'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y']
129          indices = [acid_letters.index(c) for c in list(sequence)]
130          return indices
131
132      except Exception:
133          print(sequence)
134          raise Exception
135
136  x_final = []
137  for i in range(len(x)):
138      temp_x = sequence_to_indices(x[i])
139      x_final.append(temp_x)
140
141  x_final = np.array(x_final)
142  y_final = np.array(y)
```

- Line 125-130: Amino acid sequences can't be feed as it was. They need to be converted to numerical values. "sequences_to_indices()" function does this job. Every character converted its list index (A to 1, C to 2 etc.).
- Line 137-139: Converts every element of x and adds to the new list "x_final".
- Line 141-142: Converts x and y's type into numpy array.

Training The LSTM Model

Embedding

Before training the model, we need to mention word embedding. Word embedding represents words in continous vectors, not in 0s and 1s. It differs from one hot encoding in this way. They can represent any word in few dimensions, mostly based on the number of unique words in our text. They are dense, low dimensional vectors.



Geometric relationship between words in a word embeddings can represent semantic relationship between words. Words closer to each other have a strong relation compared to words away from each other.

Vectors/words closer to each other means the cosine distance or geometric distance between them is less compared to others.

There could be vector “male to female” which represents the relation between a word and its feminine. That vector may help us in predicting “king” when “he” is used and “Queen” when she is used in the sentence.

Shapes

1.) [. . . . . . 5]   Initial shape: (5,). List has 5 elements.

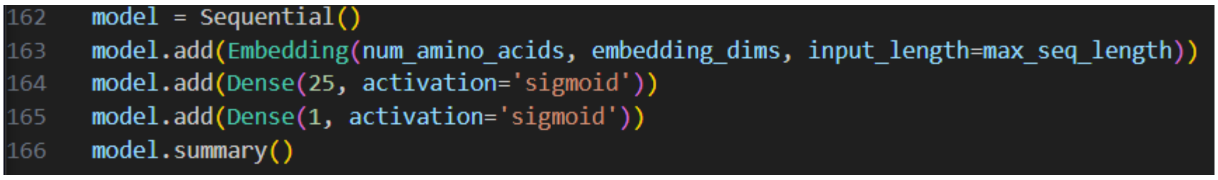
2.) [[. . . . . 500]   []   []   []   []]   Shape is now: (5, 500). 5 list each one of them have 500 elements.

3.) [[

```
 [. . . . . . . . . . 23]          (where 23 is the number of amino acids)
  .
  .
  .
  500
]
[]
[]
[]
[]
]
```

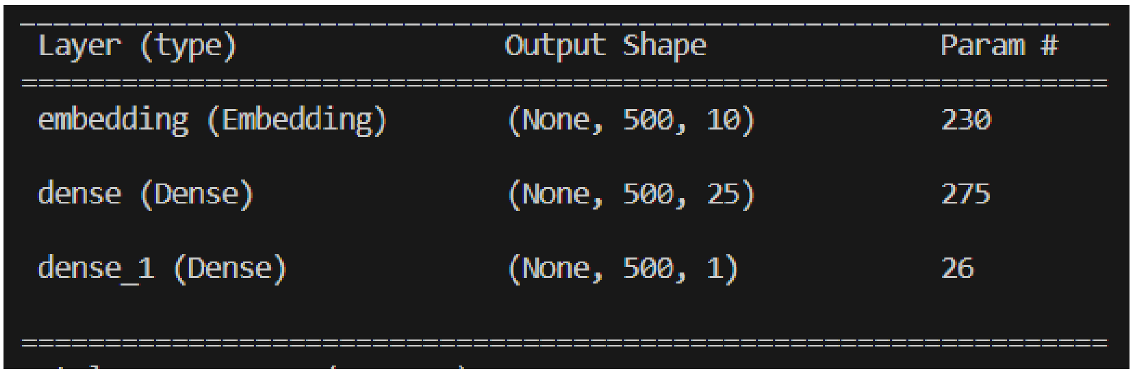
So, the final shape will be: (5, 500, 23). 5 different lists with total element length of 500 and distinct element count of 23.

Usage of Flattening



```
model = Sequential()
model.add(Embedding(num_amino_acids, embedding_dims, input_length=max_seq_length))
model.add(Dense(25, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

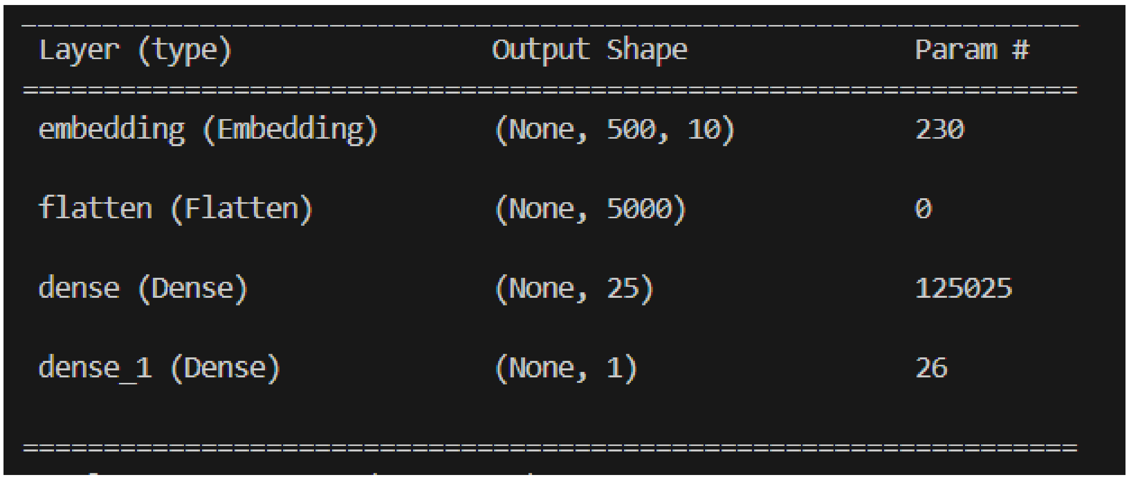
In a scenario when flattening doesn't applied, dataset is can't be fitted into the model. Lets look at the summary of the model:



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 500, 10)           230

 dense (Dense)               (None, 500, 25)           275

 dense_1 (Dense)             (None, 500, 1)            26

=================================================================
```

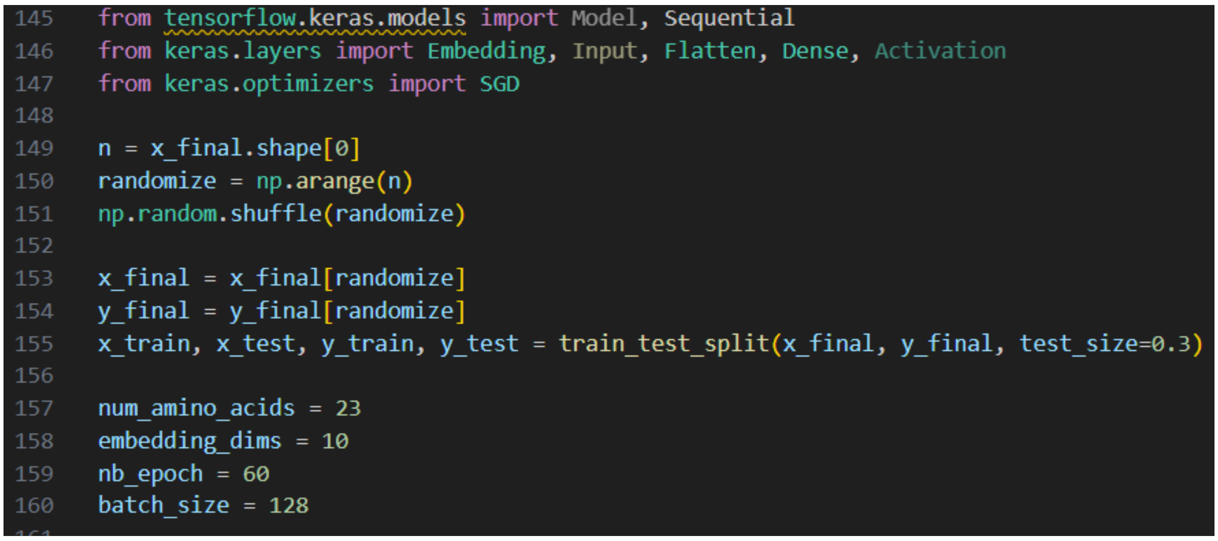
'None' is a placeholder for the batch size. It is none because batch size varies. Numerical values are the same as data.shape's. First one is row count, second one is column count.

In 3rd dense layer, it has one output for every 500 element in sequence. After flattening:



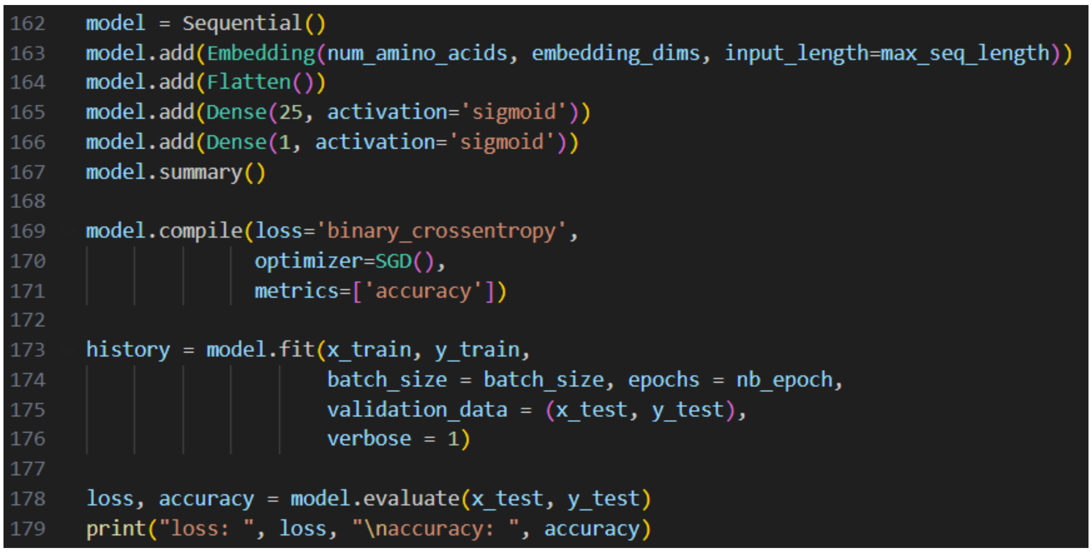
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 500, 10)           230

 flatten (Flatten)           (None, 5000)              0

 dense (Dense)               (None, 25)                125025

 dense_1 (Dense)             (None, 1)                 26

=================================================================
```

Code



```
145    from tensorflow.keras.models import Model, Sequential
146    from keras.layers import Embedding, Input, Flatten, Dense, Activation
147    from keras.optimizers import SGD
148
149    n = x_final.shape[0]
150    randomize = np.arange(n)
151    np.random.shuffle(randomize)
152
153    x_final = x_final[randomize]
154    y_final = y_final[randomize]
155    x_train, x_test, y_train, y_test = train_test_split(x_final, y_final, test_size=0.3)
156
157    num_amino_acids = 23
158    embedding_dims = 10
159    nb_epoch = 60
160    batch_size = 128
```

- Line 149: Total row count of the x list.
- Line 150: Creates a list from 0 to n-1.
- Line 151: Shuffles the list's elements.
- Line 153: Shuffles the x variables with the randomize list that we created.
- Line 154: Shuffles the y variables with the randomize list that we created.
- Line 155: Splits the x and y values into training and test sets.
- Line 157-160: Required variables for LSTM model (number of distinct amino acids, dimensions for embedding, epoch count and batch size).



```
162  model = Sequential()
163  model.add(Embedding(num_amino_acids, embedding_dims, input_length=max_seq_length))
164  model.add(Flatten())
165  model.add(Dense(25, activation='sigmoid'))
166  model.add(Dense(1, activation='sigmoid'))
167  model.summary()
168
169  model.compile(loss='binary_crossentropy',
170                optimizer=SGD(),
171                metrics=['accuracy'])
172
173  history = model.fit(x_train, y_train,
174                      batch_size = batch_size, epochs = nb_epoch,
175                      validation_data = (x_test, y_test),
176                      verbose = 1)
177
178  loss, accuracy = model.evaluate(x_test, y_test)
179  print("loss: ", loss, "\naccuracy: ", accuracy)
```

- Line 162: Defining the LSTM model.
- Line 163: Creating input layer with embedding. Takes distinct amino acid count, embedding dimensions and sequence length as a parameter.
- Line 164: Flattens the shape.
- Line 165: Adding another layer to the model.
- Line 166: Output layer to the model.
- Line 169-171: Compiles the model.

- Line 173-176: Fits the dataset into the model and saves it into variable.
- Line 178: Calculates loss and accuracy values of the model.