Lab Journal 01.04.2023

I. Exploring the dataset

Our dataset is downloaded from here (GitHub repo of the original paper).

```
data = pd.read_csv("ACE2_train_data.csv")
```

(for now we work from the same directory we download our data in).

What is the dataframe we have downloaded?

In the paper, two characteristics of mutated RBDs of COVID-19 were predicted: affinity to the human ACE2 receptor (binary: binding/non-binding, see here and later here) and antibody escape (for 4 different antibodies). `ACE2_train_data.csv` should contain the training binding information.

First we inspect the data in order to understand its structure and, insbesondere, technical details like the data delimiter. For that last task specifically we use the csv module of the standard library.

```
import csv

# read in the first ten lines of the file
with open('ACE2_train_data.csv', 'r') as f:
    first_lines = [f.readline() for _ in range(10)]

# use csv.Sniffer to determine the delimiter
dialect = csv.Sniffer().sniff('\n'.join(first_lines))

print(dialect.delimiter)
```

from which we get

```
>>,
```

Hence, when constructing our Dataset class, we need to remember to use ',' as a delimiter when reading the dataframe. Next, let's explore the dataset with the help of the standard Pandas instruments: we begin with `data.info`

```
In [41]: data.info
```

```
Out[41]: <bound method DataFrame.info of         Unnamed: 0                 junction_aa  consensus_count  Label  Distance
         0              287261  KNAGFNCYNPLETYGFWRTGGVDW                1      1         9
         1              467439  KNEQFNCYGPINAYGFQRTGGEDW                1      0        10
         2              414422  KNQKFNCYVPLFHYGFWPTVGVGF                1      1         8
         3              103144  KNQGFNCYNPLVNYGFYRTNGRSF                1      1         9
         4              478954  KNRGFNCYKPLPGYGFQRTDGINW                2      0         9
         ...               ...                       ...              ...    ...       ...
         406881          16530  KNKGFNCYIPIEDYGFQRTSGRSY                2      0         9
         406882          48280  KNEGFNCYNPITEYGFWTTSGLDW                2      1        10
         406883         420449  KNGKFNCYHPIVRYGFHPTVGRGY                2      1         9
         406884         173734  KNGQFNCYIPIAGYGFLPTLGVSY                1      0         9
         406885         554432  KNRGFNCYTPIFKYGFFTTWGRNY                1      0        10

         [406886 rows x 5 columns]>
```

Hence, the name of the columns are [no name], junction_aa, consensus_count, Label, Distance.

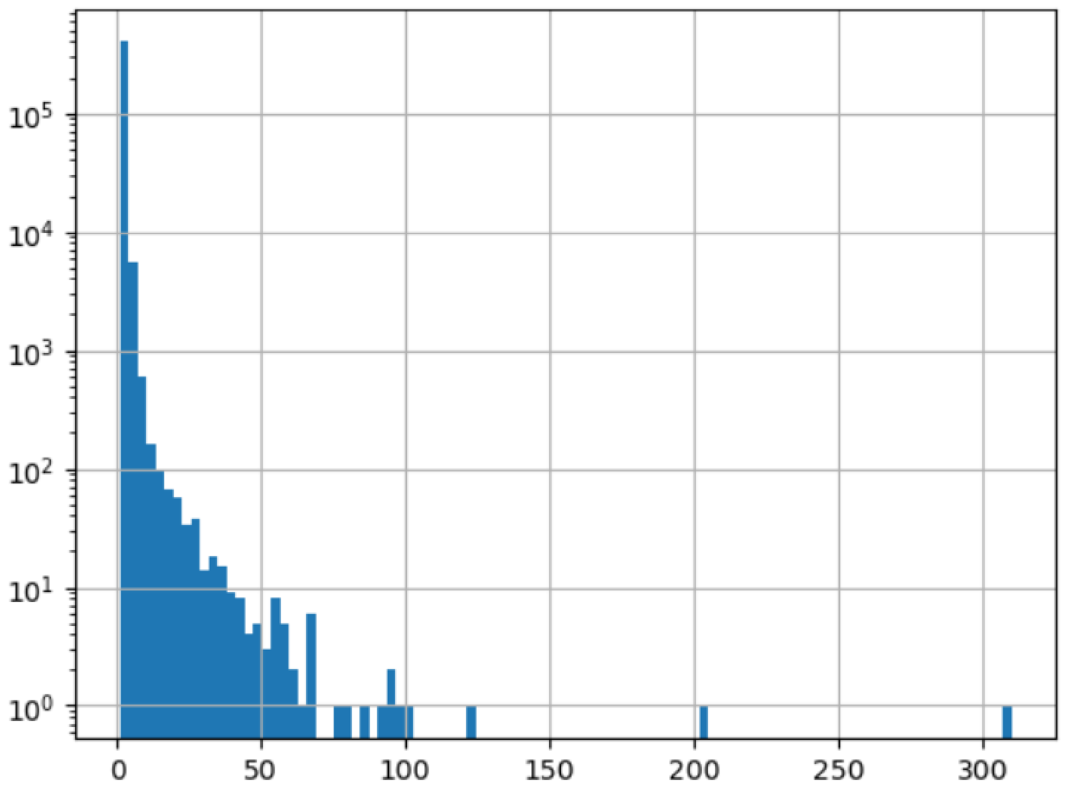
```
In [51]: data.describe()
```

Out[51]:

	Unnamed: 0	consensus_count	Label	Distance
count	406886.000000	406886.000000	406886.000000	406886.000000
mean	323195.466440	1.443908	0.500027	9.017398
std	189813.180032	1.423285	0.500001	1.167593
min	0.000000	1.000000	0.000000	1.000000
25%	154232.250000	1.000000	0.000000	8.000000
50%	325632.500000	1.000000	1.000000	9.000000
75%	483459.500000	2.000000	1.000000	10.000000
max	663081.000000	310.000000	1.000000	12.000000

What are their "physical" meanings? The unnamed column is, apparently, just an ID. Distance appears to be the Hamming distance from the reference RBD. The only binary characteristic is Label)
so it should be the binding/non-binding.
`junction_aa` is the amino acid sequence. `consensus_count` poses the most questions.
As the following histogram indicates (note the logarithmic scale),

```
In [56]: data.consensus_count.hist(bins=100, log=True)
Out[56]: <AxesSubplot: >
```



the vast majority of data has the value of `consensus_count` equal to 1-10, less then 1% has more, but the maximal value is 300. What exactly is `consensus_count`, is, unfortunately, not immediately clear from the text of the article. We ignore this parameter for now.

Let's try to build a DataLoader.

```python
class ACE2Dataset(Dataset):
    def __init__(self, csv_file):
        self.data = pd.read_csv(csv_file, sep=',', header=None)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        #if torch.is_tensor(idx):
        #    idx = idx.tolist()

        sample = {'id': self.data.iloc[idx, 0],
                  'junction_aa': self.data.iloc[idx, 1],
                  'consensus_count': self.data.iloc[idx, 2],
                  'Label': self.data.iloc[idx, 3],
                  'Distance': self.data.iloc[idx, 4]}

        return sample
```

This does not work, because we need a vector representation for the `junction_aa` parameter. Our simplest option is to use one-hot encoding. However, a better way is to use a word embedding method, or, even better, the Doc2Vec. For that, we use the gensim library.
We use the AA-sequences from the train dataset to train a word2vec embedding model:

```
from gensim.models import Word2Vec
vec_size = 10
win_size = 2
epochs = 100

sentences = data.iloc[:, 1].apply(list)

model = Word2Vec(sentences, vector_size=vec_size, window=win_size,
min_count=1)
model.train(sentences, total_examples=len(sentences), epochs=epochs)
```

and save it:

```
model.save('Word2Vec_vs10_ws2_e100.model')
```

The drawback of this method is that we only obtain embeddings for individual letters. To obtain embeddings for whole sentences, we have no better method than to just sum all these vectors and normalize, thus forgetting all structural information. A better option seems to be Doc2Vec, that provides vector embeddings for whole sentences (but ultimately, I suppose, it will end up in a CNN embedding).

```
from gensim.models import Doc2Vec
from gensim.models.doc2vec import TaggedDocument
min_count = 1
docs = data.iloc[:, 1]

documents = [TaggedDocument(words=list(data.iloc[i, 1]), tags=[str(i)])
for i in range(len(data))]

model = Doc2Vec(documents, vector_size=10, window=2, min_count=1,
workers=4)
model.save('doc2vec_vs10_ws2_mc1.model')
```

The next steps:

1. Create a one-hot embedding for reference
2. Create a CNN encoding
3. Train a CNN/RNN based on this embedding. Compare the results for different encodings and choose the best. Document the time of work and quality metrics

on `ACE2` dataset.

4. Use these embeddings and dataloaders to train a CNN/RNN on one of the four antibody escape problems.

TBR: Word2Vec Tutorial:
https://www.kaggle.com/code/pierremegret/gensim-word2vec-tutorial