# CS4650 Neural Predictions for Year of Authorship of Historical Texts

May 2, 2023

## 1 Download Data

1. Download the pickled gzip file from google drive.
2. Check the filehash
3. Unpickle and read
4. Check that there are `5051` total books

[1]:
```
import pickle
import gzip
import gdown
import hashlib

import math
import time
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
from torch.nn.utils.rnn import pad_sequence
import torch.nn.functional as F
from torch import optim

from functools import partial

import numpy as np
import pandas as pd
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
```

[2]:
```
!pip install -q transformers
!pip install -q pytictoc
from transformers import get_linear_schedule_with_warmup
from pytictoc import TicToc
```

[3]:
```
# Download from public url in google drive:
```

```
# https://drive.google.com/file/d/1MzHuNU0RzjYILE1-dnsejpn6rLtX27b1/view?
↪usp=sharing

! gdown 1MzHuNU0RzjYILE1-dnsejpn6rLtX27b1
```

```
Downloading…
From: https://drive.google.com/uc?id=1MzHuNU0RzjYILE1-dnsejpn6rLtX27b1
To: /content/FULL.gz
100% 1.34G/1.34G [00:04<00:00, 287MB/s]
```

[4]:
```
# Check file-hash to verify download
with open('FULL.gz', 'rb') as file:
  print("Reading gzip file...")
  contents = file.read()

print("Checking file-hash...")
hashobj = hashlib.sha256(contents)
correct = "33cc853c499d08b841be17517472b0ffb703235d8239c198f5d5ad5160a5a11f"

print(f"sha256: {hashobj.hexdigest()}")
print(f"Hash matches: {str(str(hashobj.hexdigest()) == correct).upper()}")
```

```
Reading gzip file…
Checking file-hash…
sha256: 33cc853c499d08b841be17517472b0ffb703235d8239c198f5d5ad5160a5a11f
Hash matches: TRUE
```

[5]:
```
# Read/Expand the data
import random
print("Opening file...")
with gzip.open('FULL.gz', 'rb') as file:
  print("Expanding file...")
  all_books = pickle.load(file)

  # Only use a random subset of all the books
  random.shuffle(all_books)

print("DONE!")
```

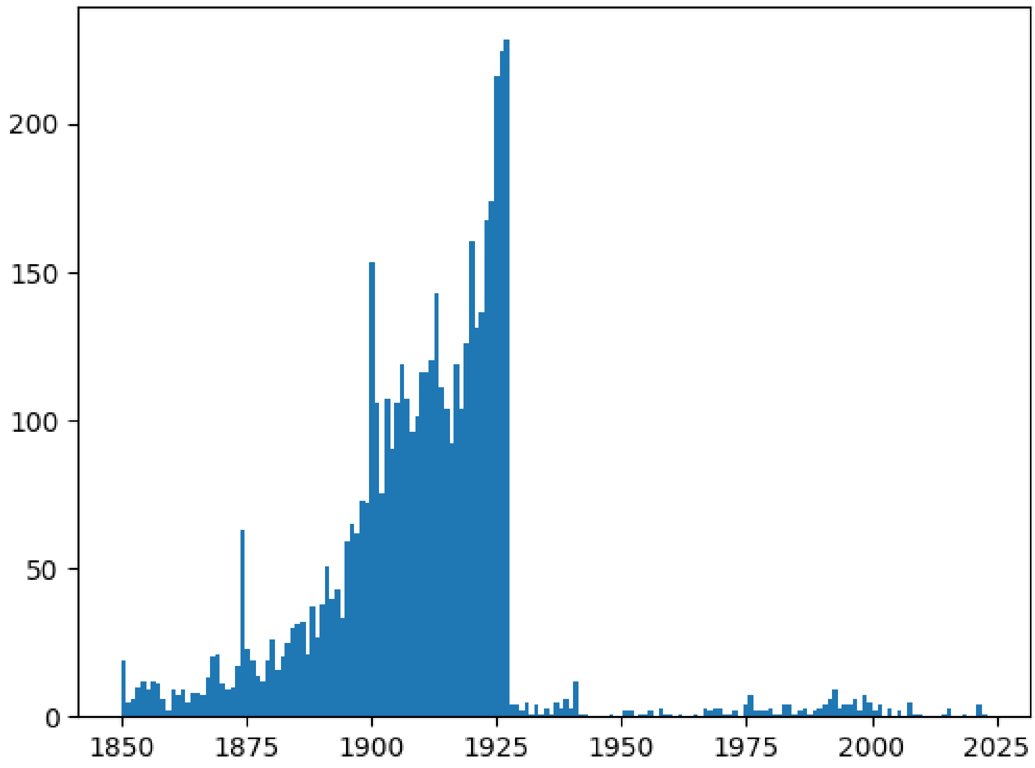
```
Opening file…
Expanding file…
DONE!
```

[6]:
```
print(f"Total number of books: {len(all_books)}")
print(f"      Have all books?: {len(all_books) == 5051}")
```

```
Total number of books: 5051
      Have all books?: True
```

[7]:

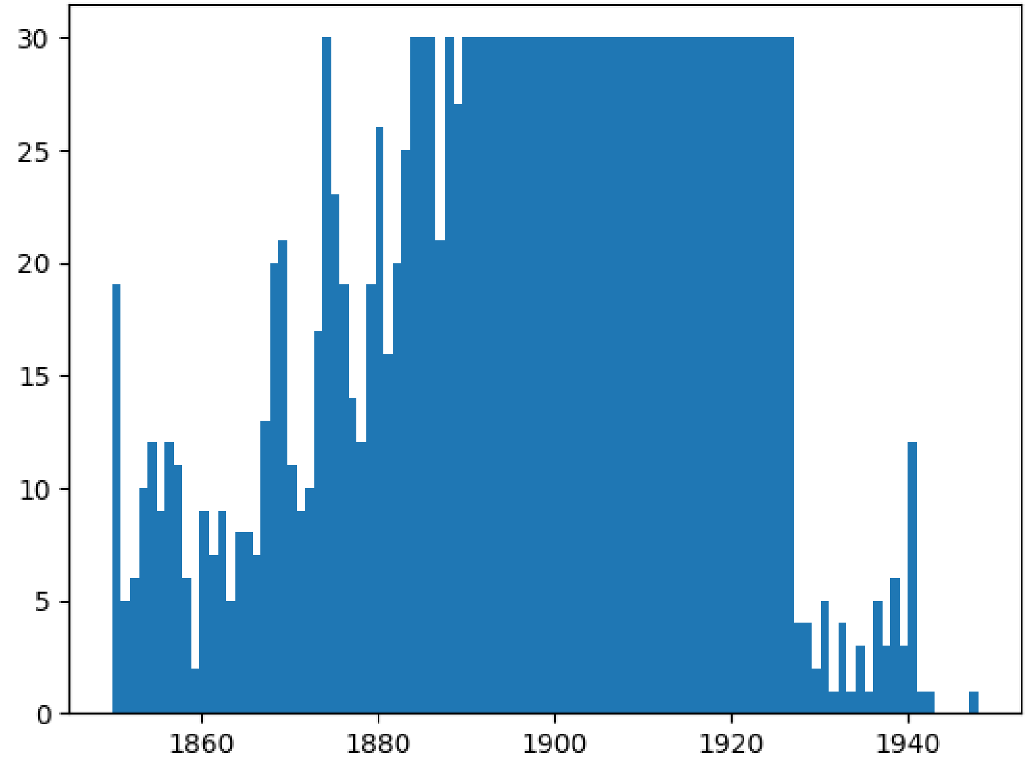
```python
all_years = [b['year'] for b in all_books]
plt.hist(all_years, bins=max(all_years)-min(all_years)+1)
plt.show()
```




[8]:

```python
# Cap books by year and reduce down to 1850-1950
subsetbooks = []
counts = {}
CAP = 30
subsetyears = []
for book in all_books:
  y = book['year']
  if counts.get(y, 0) >= CAP or y > 1950:
    continue
  else:
    subsetbooks.append(book)
    subsetyears.append(y)
    counts[y] = counts.get(y, 0)+1

books = subsetbooks
print(f'total num of books: {len(books):,}')
plt.hist(subsetyears, bins=max(subsetyears)-min(subsetyears)+1)
plt.show()
```

```
total num of books: 1,815
```




[9]:
```python
# Reduce number of books and attempt normal distribution of years
all_years = []
for book in books:
  all_years.append(book['year'])

GLOBAL_MIN_YEAR = min(all_years)
GLOBAL_MAX_YEAR = max(all_years)
GLOBAL_YEAR_RANGE = GLOBAL_MAX_YEAR - GLOBAL_MIN_YEAR + 1

print(GLOBAL_MIN_YEAR, GLOBAL_MAX_YEAR)
print(GLOBAL_YEAR_RANGE)
```

```
1850 1948
99
```

### 1.1 Quick data demo

Each item in the `books` list has keys: - `identifier`: The arbitrary id assigned by The Internet Archive for the book (useful if you want to find the original book as a pdf on IA). - `creator`: The author or publisher of the book. - `year`: Year of writing (according to IA). This should be our target attribute. - `title`: The full title of the book. - `text`: The full plain-text of the book, straight from the `.txt` files found on IA. - NOTE: We will probably want to make sure to ignore the first few pages of tokens in the texts because most have the page with the date of publication, which would just be cheating. - Also there's been no cleaning in regards to language or special characters.

[10]:
```
# Show info available on a random book (the first one in the set)
b = books[0]
for key, val in b.items():
  if key != 'text':
    print(f"{key:>10} | {val}")
  else:
    short = val[200:300].replace('\n', ' ').strip()
    print(f"{key:>10} | {short}")
```

```
identifier | apocryphatransla0000unse_c2i5
      year | 1914
     title | The Apocrypha : translated out of the Greek and Latin tongues ;
being the version set fourth A.D. 1611, compared with the most ancient
authorities and revised A.D. 1894
      text | , aad be    4    ; abe    7 oe            i  * a \  a) ; ¢  ter "
it Se a .  ¢ ¥ , . F  . s he 4
```

## 2 Creating Dataloader

Current dataset just has plain text, no cleaning at all.

The first few pages of most books will have publication year, we will need to be sure that we are removing these first few pages before passing text into the model.

We also should try to split the data down to paragraphs to avoid passing in full books which would be excessive.

[11]:
```
# Parse Books to make paragraphs

total_words = 0
ids = []
years = []
texts = []

max_words = 0
max_text = None

for book in tqdm(books):
  # Filter for longer paragraphs and exclude first pages
  for i, para in enumerate(book['text'].strip().split('\n\n')):
    if (len(para) < 200 or len(para) > 3_000) or (i <= 10):
      continue

    num_words = len(para.strip().split(' '))
```

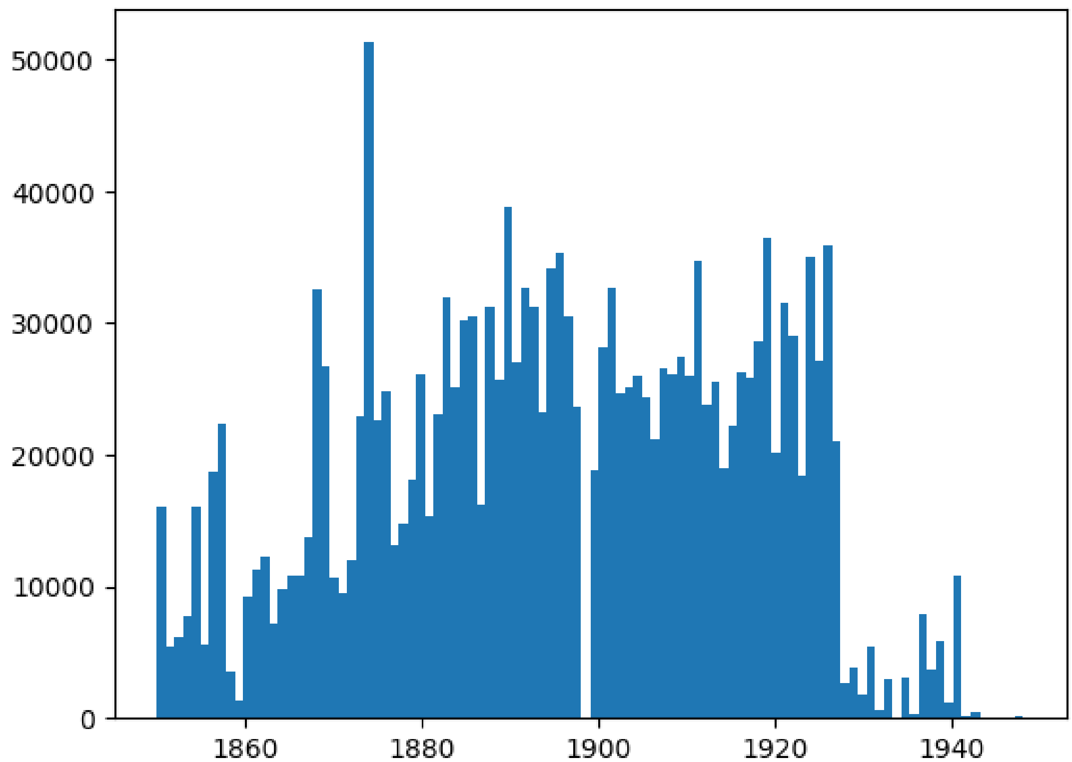
```python
    total_words += num_words
    if num_words > max_words:
      max_words = num_words
      max_text = para


    ids.append(book['identifier'])
    years.append(book['year'])
    texts.append(para.strip())

para_df = pd.DataFrame({
    'ids': ids,
    'year': years,
    'text': texts,
})

print(f'       Total paras: {len(texts):,}')
print(f'Total num of words: {total_words:,}')
print(f'    Average length: {total_words / len(texts):.1f}\n')
print(para_df)

print(f'BUILDING PARAGRAPH LIST...')
para_df_list = list(para_df.iterrows())
```

```
  0%|          | 0/1815 [00:00<?, ?it/s]

       Total paras: 1,797,845
Total num of words: 224,245,765
    Average length: 124.7

                                  ids  year  \
0        apocryphatransla0000unse_c2i5  1914
1        apocryphatransla0000unse_c2i5  1914
2        apocryphatransla0000unse_c2i5  1914
3        apocryphatransla0000unse_c2i5  1914
4        apocryphatransla0000unse_c2i5  1914
…                                   …   …
1797840       jahrbucherfurpro0002unse  1875
1797841       jahrbucherfurpro0002unse  1875
1797842       jahrbucherfurpro0002unse  1875
1797843       jahrbucherfurpro0002unse  1875
1797844       jahrbucherfurpro0002unse  1875

                                                      text
0        The present work, owing to various circumstanc…
1        The Revision of the Authorised Version of the …
2        It was resolved (March 21, 1879) that, after t…
3        The London Committee was to consist of the fol…
```

```
4        The Westminster Committee was to consist of th…
…                                                      …
1797840  Dagegen werden wir in dem Maassstab, den ‚Jesu…
1797841  „Satz vom zureichenden Grunde" des Lohns. Frei…
1797842  göttlichen Gnade wollen wir hier um so weniger…
1797843  Worte kommen lässt. Und gerade dem berufsmässi…
1797844  in dem Wörtchen versteckt liegt, in dem sie fr…

[1797845 rows x 3 columns]
BUILDING PARAGRAPH LIST…
```

[12]:

```
# Get year distribution BEFORE subselection
plt.hist(para_df.year, bins=GLOBAL_YEAR_RANGE+1)
plt.show()
```




[13]:

```
# TOTAL_PARA = 3_200
TOTAL_PARA = 2_200
# TOTAL_PARA = 1_200
allotment = (TOTAL_PARA // GLOBAL_YEAR_RANGE) + 0
print(f'Allotment is: {allotment}')
CUTOFF = len(para_df) // 5
print(f'CUTOFF: {CUTOFF}')
```

```python
print(f'Total number of available para: {len(para_df_list)}')

used = []
counts = {}
# for i, (idx, row) in tqdm(enumerate(random.sample(para_df_list, CUTOFF))):
for i, (idx, row) in tqdm(enumerate(random.sample(para_df_list,␣
↪len(para_df_list)))):
  if i >= len(para_df_list)-1:
    print(f'CHECKED ALL PARAS')

  y = row.year
  currcount = counts.get(y, 0)
  if currcount >= allotment:
    continue
  else:
    counts[y] = currcount + 1
    used.append(idx)

  if len(used) >= TOTAL_PARA:
    print(f'BREAKING because total used: {len(used)} is more than TOTAL_PARA:␣
↪{TOTAL_PARA}')
    break



print(used)
subset = para_df.iloc[used]
print(subset)
print(len(subset))
```

```
Allotment is: 22
CUTOFF: 359569
Total number of available para: 1797845
```

```
0it [00:00, ?it/s]
```

```
CHECKED ALL PARAS
[436266, 463624, 1749883, 1496059, 589264, 1137727, 974481, 620776, 1718188,
72325, 208905, 400618, 1528618, 289330, 482067, 587597, 497157, 923836, 467313,
627712, 1783891, 95643, 723107, 1779550, 1184351, 333430, 508709, 439101,
717874, 461309, 1171450, 896821, 454247, 378011, 1439831, 374290, 712162,
1628307, 184713, 1029570, 1299849, 503122, 667269, 826398, 279980, 1612251,
1880, 795759, 602707, 1182024, 995186, 95034, 751503, 399876, 1326030, 1142800,
952421, 555268, 1533578, 561662, 1779246, 893294, 658062, 290372, 607900,
1708335, 1451354, 1558645, 409426, 517590, 1161280, 341447, 1296776, 892266,
60871, 361208, 1121260, 1737445, 1036240, 152242, 1138026, 103948, 708615,
383606, 1611128, 1597777, 620634, 1207413, 1151691, 1726978, 1443652, 966453,
366423, 252433, 783613, 516941, 1673546, 1719242, 1759538, 95151, 1547028,
```

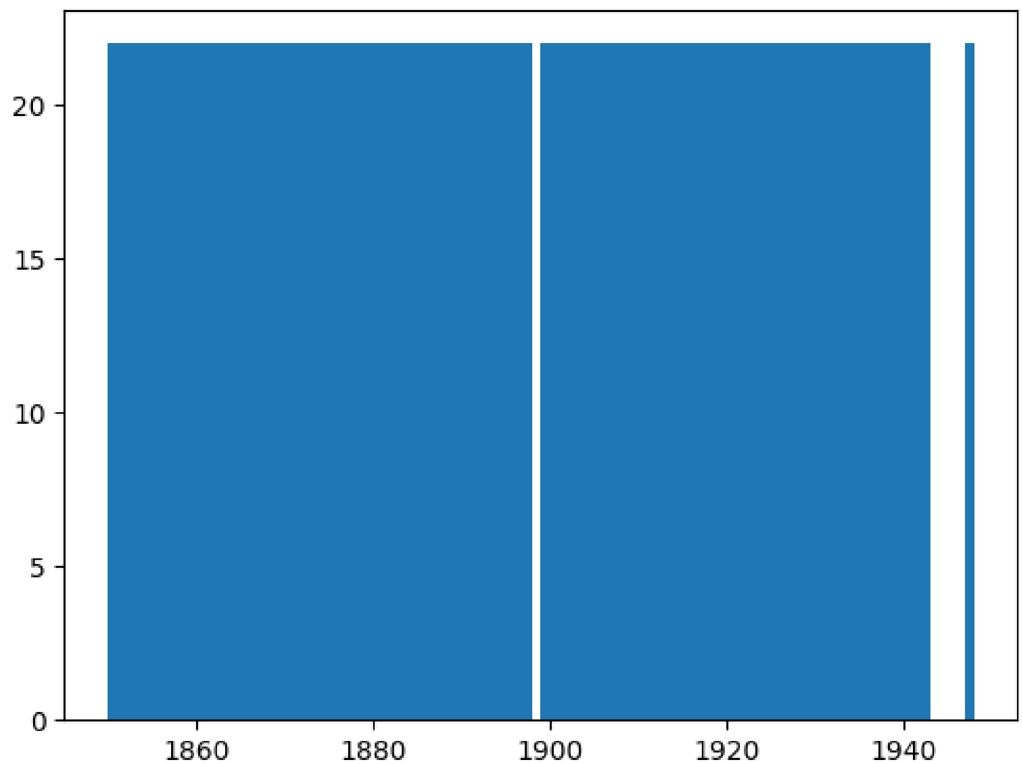
1387374, 226054, 899754, 1509373, 306047, 78046, 864499, 1718225, 1750163, 1382847, 146222, 1769617, 1786810, 1054826, 1032808, 491664, 1266627, 198908, 1388243, 1755556, 991889, 1735361, 621304, 224597, 1149287, 253045, 1172655, 293475, 679920, 1496292, 1114162, 1774344, 1655008, 63209, 350132, 416238, 1279568, 911024, 798942, 603403, 1242716, 945955, 830731, 1435952, 1588906, 869168, 1151445, 616603, 752276, 66885, 370711, 1189983, 218559, 164399, 595067, 255369, 350673, 1769103, 21099, 759076, 1383131, 1035363, 1033241, 174115, 814726, 1557930, 1579502, 643999, 1374547, 465847, 1733660, 954469, 1766126, 191916, 1070675, 388257, 535909, 1436237, 724336, 901417, 368906, 1774939, 1378432, 1595886, 1459560, 277282, 1591605, 288030, 310475, 805333, 1629304, 1710103, 68272, 1458338, 529353, 241602, 484880, 1768693, 1669795, 459574, 1196895, 120885, 822530, 799074, 1668936, 723599, 238451, 692812, 1223673, 1120150, 1084524, 1567366, 974106, 881234, 1093208, 595517, 53982, 1329450, 1724460, 1587006, 584182, 748815, 919753, 40079, 1195511, 953326, 619809, 84665, 842353, 816208, 1250753, 1492966, 719499, 565441, 1033696, 504331, 1441729, 1385877, 1342356, 805264, 738812, 1298129, 553762, 864090, 790276, 1614257, 1094468, 1162210, 166473, 914390, 342571, 1247955, 459605, 1571236, 1023544, 1313185, 1577430, 623587, 194371, 894313, 76878, 1127927, 960108, 1789034, 252926, 1031980, 1600204, 1704265, 671707, 610593, 276820, 520055, 638146, 1355436, 1719450, 158540, 1435181, 32824, 671164, 32273, 414842, 1523867, 1216669, 179146, 1220683, 1402990, 1646943, 1335554, 950504, 1443494, 880753, 573355, 180430, 1328985, 827678, 1619480, 568229, 543979, 1509992, 1100782, 1469952, 93391, 1797586, 963068, 352496, 1090534, 35118, 81809, 1132050, 1513167, 204273, 1141609, 919764, 648690, 836958, 1398050, 735333, 823252, 880505, 463970, 1143046, 1360893, 1065783, 1516469, 213585, 658275, 829844, 764073, 1194504, 1534355, 808073, 941037, 1746373, 1422983, 1382234, 1780959, 402764, 1764883, 210677, 235705, 1630525, 412682, 1329006, 923385, 1634829, 949794, 621405, 1389124, 1447986, 1621739, 1645776, 565452, 1377060, 289942, 910965, 11058, 1394153, 215934, 1054812, 337613, 1344334, 993606, 572354, 595741, 709592, 1342143, 1315096, 1320440, 241454, 140701, 1572771, 116389, 1566132, 1628230, 1592798, 1622047, 773939, 134824, 727326, 1483138, 1511081, 1136567, 1508285, 1253024, 987485, 793875, 1345368, 1244620, 1350640, 672642, 568527, 570811, 99411, 1704703, 1733130, 918541, 457775, 398390, 247600, 11162, 1483893, 1202197, 353086, 1783939, 1658829, 126512, 843907, 340915, 1717079, 1367885, 434637, 45032, 594106, 1006740, 1330991, 581358, 1743807, 5328, 1077192, 1747736, 506085, 1342965, 431445, 1493769, 1188231, 1640135, 402876, 315861, 644662, 746740, 1022616, 1153141, 379047, 370718, 1139634, 261045, 1186286, 969994, 1484720, 1297003, 670868, 1543593, 989959, 446749, 1593872, 42457, 1598280, 1017526, 188103, 284931, 39197, 736612, 1014728, 1538592, 233615, 785220, 1147750, 1101000, 455779, 462805, 842468, 1793084, 545540, 1767918, 206649, 795462, 876085, 728419, 1208264, 98897, 1576439, 1560863, 1594375, 336059, 1221369, 1038021, 785808, 914420, 1097118, 271051, 131984, 1126487, 1782217, 1504298, 237440, 1626725, 1575426, 1429222, 848080, 71374, 633349, 1785231, 271448, 734497, 1003536, 1269478, 1559791, 1046777, 1181947, 220439, 1225523, 729167, 1116245, 886967, 126611, 721275, 1715265, 605143, 139845, 583185, 1176962, 1486493, 113772, 953201, 1459413, 67274, 427081, 428688, 76419, 1045527, 593013, 528313, 1328891, 1513060, 928200, 1271207, 313399, 1281059, 257746, 245097, 652844, 625080, 1221856, 1614647, 1517847, 227353, 1595080, 337706, 706711, 552072, 1261654, 1313694, 1343316, 1621083, 1020162, 480873, 1323095, 1168405, 273470, 1533582, 932928, 1483590, 1169923, 177472, 1079060, 1408159, 1548507, 44445, 1669722, 488180, 110457, 1027406, 1181122, 1306589, 577524, 1473309, 140586, 1521765, 51412, 793648, 422259, 1649025, 956619, 213219, 803236, 921598, 804845, 10812, 824494, 214985, 672432, 1489896, 777944, 1670281, 479187, 502370, 160005, 1512152, 1509051, 1786990, 527276, 28160, 1036895, 1114562, 234123, 390433, 959770, 1575106, 1407603, 1046395, 1647448, 1499763, 257798, 1474457, 3411, 532901, 1633573, 441052, 1233116, 660864, 749495, 112236, 1149126, 13370, 314076, 946067, 1261255, 1666229, 534212, 866835, 674102, 1482797, 493997, 618103, 1679813, 1315939, 538534, 265684, 1088172, 1190515, 807901, 1261773, 1069518, 864657, 1471892, 774313, 516413, 73416, 985311, 1284489, 1242245, 140497, 389634, 1420235, 1679502, 513815, 1696408, 210012, 49688, 1043584, 171787, 1340730, 1010165, 553538, 233568, 1640868, 922168, 605929, 1396962, 706039, 488373, 1565172, 124515, 946253, 1721112, 1019784, 571090, 1488748, 76145, 1784808, 1510388, 1181078, 1059667, 1379859, 452167, 703747, 1172902, 644421, 926190, 627996, 95984, 5381, 832120, 1707661, 205665, 1397104, 837733, 434556, 848142, 1274007, 1484459, 926628, 741388, 561013, 1174222, 46300, 232545, 1636735, 499367, 1350305, 1060998, 21347, 222789, 1180108, 1730034, 1330756, 826686, 212450, 354773, 608697, 1278236, 1318204, 1306327, 1158857, 280534, 1337996, 61146, 186612, 582629, 941767, 1360297, 800621, 815328, 531580, 1388001, 160151, 533575, 320575, 985492, 1100062, 1562875, 286863, 1045949, 254241, 1327638, 1341292, 194180, 1784366, 513803, 977602, 52759, 127654, 135643, 1713818, 259565, 704809, 1011601, 722015, 689282, 419312, 1035777, 828119, 1425465, 880663, 1518369, 775743, 1149778, 917988, 717938, 1050569, 1667150, 191697, 1581758, 834305, 1481322, 655719, 1097671, 488655, 554055, 1753222, 672114, 1551099, 787092, 653555, 1644233, 345726, 1153454, 582164, 1402131, 1448387, 323781, 1414668, 1651068, 187012, 7308, 194063, 411839, 124062, 1361482, 612777, 183978, 504401, 29991, 940527, 1357337, 1143523, 1528440, 1678569, 1666695, 1137335, 1688053, 1679684, 197474, 1520442, 32042, 1742084, 1296093, 860133, 1557719, 933137, 1152030, 1360039, 1170425, 135818, 1654389, 678011, 1053224, 1237842, 323409, 124170, 125621, 205145, 1777073, 752430, 986490, 119131, 922474, 1004671, 935524, 170948, 677530, 1216278, 895816, 705424, 1166500, 1195772, 882948, 1491105, 823737, 113032, 1296827, 838873, 711848, 897315, 1290862, 512271, 493821, 576962, 1208587, 2286, 743385, 1196754, 1761719, 1515701, 1650785, 628795, 540951, 1057991, 1413608, 783481, 242902, 1721444, 1544828, 752282, 1283468, 39210, 567591, 1787710, 30240, 396947, 584445, 125342, 358655, 248340, 1570669, 439135, 307914, 955816, 1033489, 342410, 1742514, 1269244, 1742355, 874827, 1733170, 850799, 1488760, 1301167, 774460, 407469, 1261746, 281233, 1040277, 1780166, 1666173, 720300, 1514334, 127850, 959688, 1701371, 224042, 1215426, 42973, 1669487, 863633, 770004, 550782, 86604, 1001435, 893524, 1117901, 1709624, 1546616, 127938, 1789826, 732618, 23170, 1632425, 1099032, 1711695, 728490, 1713693, 1652220, 933531, 847283, 1423100, 118592, 1782970, 485537, 1327522, 1088933, 1614365, 697871, 1782132, 1528148, 237471, 560305, 688118, 802890, 1265298, 880892, 1527843, 792014, 1349580, 310157, 906934, 627478, 483315, 1362119, 576984, 922774, 824379, 988831, 896922, 1153092, 871979, 1029837, 836942, 1783107, 332605, 1530275, 893157, 490597, 868074, 933300, 397326, 1588955, 1551393, 1779892, 1178637, 658719, 527572, 1042996, 1723039, 195149, 1671775, 1616110, 826531, 1768944, 445171, 1750886, 439942, 777098, 1169772, 457560, 1200436, 990380, 1780679, 133645, 586913, 1349147, 114575, 1785833, 892764, 670586, 795298, 1412874, 503229, 1631061, 515988, 1162266, 183666, 1240501, 368370, 1363549, 1267495, 1597613, 714400, 1330964, 1559857, 288607, 742084, 591744, 404981, 1741298, 1309287, 1478328, 1638578, 1328455, 677109, 528030, 89938, 454014, 528893, 1075840, 121311, 1005286, 577502, 208594, 1324284, 191633, 692391, 1599712, 1063759, 86370, 267684, 1647208, 1308194, 1218462, 381149, 1115244, 536449, 1710972, 841087, 712439, 1268800, 740200, 928806, 571117, 1008989, 1065167, 553630, 927907, 1188087, 1125769, 1281096, 517641, 839757, 1079044, 107704, 577746, 1567734, 953055, 1695662, 1236824, 213603, 1490882, 1070505, 687478, 558895, 544095, 1185058, 1120859, 531504, 1393869, 942409, 371964, 1069325, 1029939, 143806, 729908, 319247, 1586769, 1018221, 66474, 64933, 714376, 1523333, 548948, 1739345, 1309670, 1509660, 328090, 1049849, 977484, 1171351, 791052, 1569415, 291104, 1625336, 1432028, 1456382, 1128176, 1069099, 511194, 1563285, 314190, 1187752, 462452, 661732, 945776, 498720, 1300745, 1529942, 901410, 1594813, 584491, 636602, 756351, 1258127, 951039, 1156821, 460524, 252428, 1508626, 77761, 1079825, 1260100, 1451675, 1014856, 584956, 760182, 1743637, 1652031, 1407527, 1781753, 231789, 143989, 708694, 1284771, 1292813, 280187, 361679, 1561404, 504444, 372327, 812385, 524934, 280854, 424414, 1706391, 1264907, 944794, 903874, 1036177, 1007886, 748897, 64655, 1492746, 359704, 560261, 154287, 1742525, 17423, 483090, 946269, 67507, 1401920, 1548461, 1271478, 231879, 1797784, 1393120, 1220033, 1703321, 648947, 1154041, 401949, 613482, 607736, 963720, 1172804, 1182371, 1561592, 1394751, 1314076, 110402, 197793, 1625133, 683282, 721623, 812269, 490640, 1296819, 1220022, 165028, 1491594, 1011617, 1008715, 1659972, 1292884, 1542958, 330648, 691237, 6616, 457906, 569101, 172258, 866923, 338629, 509782, 737052, 421271, 1191970, 1731632, 181980, 1664069, 327771, 298530, 1092256, 930380, 99312, 143143, 1211776, 1567575, 165268, 284507, 1698137, 1306955, 679040, 216052, 289857, 1298482, 817596, 661059, 1196543, 492322, 196283, 1318088, 17639, 836619, 1502750, 1271248, 146356, 734987, 1467147, 1708056, 1624151, 868079, 1168475, 503013, 1795397, 1662025, 1049252, 1540302, 408122, 180495, 307615, 992332, 618576, 682977, 689054, 840532, 419689, 1444522, 1051885, 646556, 1411348, 1753282, 1513191, 172587, 792077, 1023347, 677192, 1093262, 1667536, 867548, 643292, 722151, 43947, 1410230, 726204, 840595, 582894, 1701473, 726243, 1426438, 1555934, 1334200, 1540577, 697132, 1048800, 1709356, 794444, 1146161, 761636, 1449996, 684066, 178883, 1320187, 1737507, 742387, 1057923, 1549737, 1732189, 1435146, 145683, 1075314, 65437, 1093810, 1789422, 182376, 1299320, 98394, 883229, 263456, 154760, 434934, 1207780, 542912, 1191016, 1741956, 508641, 1115156, 1567658, 900979, 413532, 1252444, 1469193, 821476, 623092, 38685, 142362, 1745699, 1714510, 1568481, 1068477, 373864, 1351778, 1592362, 152341, 919524, 1701388, 692022, 1622051, 429256, 1649616, 1335025, 1620797, 1609701, 330081, 727045, 1345167, 1437964, 231340, 1521138, 915360, 1011646, 1602632, 1509457, 1512066, 249819, 938206, 523202, 1723181, 909652, 1708592, 1468220, 376021, 495109, 447095, 1695092, 1495741, 17986, 1547083, 560641, 227882, 315057, 756119, 1247059, 1742932, 38215, 1312963, 1492531, 1461729, 1027038, 397342, 76632, 1723618, 340759, 521246, 1445512, 1333187, 62354, 756395, 901720, 121672, 813156, 632318, 1720866, 1537290, 152001, 50731, 1092236, 1647421, 1313321, 64749, 1024277, 1448033, 1609182, 800623, 1258291, 148728, 151883, 251122, 1729974, 1751842, 559940, 173755, 1633034, 1575987, 714261, 1246848, 1307216, 1209486, 389789, 189914, 1249411, 1631672, 1543129, 1397392, 1771895, 523238, 331052, 1469033, 1149313, 727116, 1335523, 1264827, 1413632, 902379, 1395759, 1460976, 588094, 1754153, 521679, 79993, 1605795, 1466966, 523188, 1549404, 1653473, 1538831, 1275942, 1154759, 1482306, 1232763, 623462, 1148955, 976432, 1659358, 591517, 666923, 521771, 951433, 1620546, 577161, 1601097, 1297119, 1788445, 1788443, 1731497, 677086, 1379806, 657675, 1731804, 1673536, 1247031, 1092803, 1370333, 498527, 41666, 144491, 1588943, 1333925, 1722841, 493654, 1721931, 558214, 721038, 1346391, 754797, 1539399, 678748, 1743877, 250689, 1512422, 1638874, 1649063, 817642, 1669382, 450563, 3911, 4498, 1566799, 1546743, 674944, 734709, 1482256, 1647523, 1049667, 1380286, 1697813, 193015, 877542, 391853, 951902, 1636197, 623340, 1318141, 1720288, 1128334, 1370457, 563649, 1432192, 691557, 155453, 1788169, 81545, 1538540, 1569116, 1774568, 1049814, 857051, 1594656, 1324469, 500397, 1517163, 1206156, 1532780, 338569, 1322620, 525710, 329693, 1348259, 562019, 231916, 396974, 1736277, 1336881, 1258296, 944402, 1306636, 1394205, 660708, 1467342, 1659700, 1508368, 1672279, 517377, 1399581, 1128402, 189836, 1486625, 397292, 1663480, 1367831, 1205906, 1209520, 172835, 1797475, 169123, 1699311, 1720045, 199249, 1489511, 1759080, 1710137, 1656121, 1706987, 233063, 662453, 522567, 1742597, 1212276, 1595106, 1667630, 502509, 191372, 1788916, 1398350, 1648408, 1398613, 1416194, 30852, 679327, 1257741, 1337643, 999003, 348573, 1703027, 857272, 1532029, 1427502, 627884, 1750354, 670909, 1603830, 1743527, 1335443, 955180, 503218, 1758553, 1751853, 1757674, 893341, 502844, 1650080, 1667753, 804612, 1650798, 1265574, 499146, 1546418, 331116, 1444347, 1114844, 855776, 799739, 1145096, 1442728, 231454, 1614806, 949537, 1154741, 999201, 1348169, 1506990, 1698133, 1706880, 1619261, 1498382, 1609709, 1312384, 1316625, 798991, 1711628, 1241624, 1673071, 1445561, 498253, 1378859, 1512603, 790164, 670708, 1397258, 1600826, 1540690, 498481, 1367407, 1408210, 1758605, 1723249, 30893, 1517054, 1663404, 1292763, 827055, 1750909, 855674, 1663038, 1692117, 1700264, 1512750, 1205703, 1168356, 1788721, 1604152, 145189, 1282183, 233038, 1265155, 1742182, 1772451, 1648705, 41801, 1461733, 1020387, 1699613, 1758436, 1737147, 671500, 1570315, 1481162, 1774031, 1205730, 1557149, 1662342, 1319877, 1758096, 1370157, 1516643, 330265, 1765087, 734868, 1594975, 1202797, 1488139, 1772973, 1443659, 1525461, 356195, 1741449, 1745800, 1764285, 1149376, 200472, 1152050, 1511965, 1307585, 856222, 691163, 1758824, 1763875, 1583005, 1486868, 1511570, 1662142, 501572, 1709079, 66101, 1467137, 952740, 355891, 1557091, 857320, 1663263, 778967, 1556254, 804185, 1603216, 1785540, 523167, 1667112, 627479, 84023, 1706261, 1481704, 1480874, 1148696, 1700900, 1316455, 1746484, 1691813, 1482325, 1306973, 1552102, 1707626, 1257343, 1625414, 1624280, 1480768, 1735839, 915604, 735342, 1162089, 83777, 1241467, 735499, 330747, 856012, 1619545, 1552748, 1742275, 1750369, 1662057, 1583892, 1614208, 1487677, 1662376, 1367825, 331168, 83475, 1322506, 1686642, 1335148, 356258, 1055633, 1662349, 1322494, 1648995, 799075, 1548225, 1662402, 1637720, 827124, 1322563, 1335973, 200161, 1706416, 1335803, 855788, 1708344, 1742408, 65632, 1707725, 65365, 1686293, 1746277, 1487952, 330757, 1706070, 1206973, 1517445, 1055943, 1322502, 1249575, 65639, 1552493, 1307198, 1619894, 1487071, 1750663, 1185243, 1486305, 1746120, 1501067, 713385, 1699648, 171755, 1487340, 57274, 199606, 1648066, 1709325, 1486350, 1525988, 1517191,

```
1257405, 712844, 171352, 1682847, 1619858, 171019, 855810, 1683211, 1397455,
1398204, 1257432, 171749, 1749403, 1546028, 83979, 1699395, 1583150, 1307439,
65353, 1335293, 57607, 1320175, 1249574, 1683044, 1648946, 1648435, 1486665,
1552991, 1015534, 1647568, 1745976, 66209, 1546765, 1552006, 1335547, 1682994,
1750428, 1015470, 1517568, 1685582, 1619136, 1043250, 66193, 1683184, 713197,
1415848, 1397930, 1552774, 1517073, 713388, 713020, 330841, 1397653, 1316524,
1614872, 1552402, 1686188, 713225, 329821, 713198, 1552327, 1633663, 1524127,
1787988, 1306843, 330714, 1249498, 1485966, 1486061, 1487206, 952641, 1546910,
1043172, 1486966, 1487696, 778986, 1487233, 1457151, 1686932, 1306892, 1307516,
1682855, 1686440, 1306844, 1683171, 1614784, 1517542, 1457254, 778787, 1316726,
1517589, 65770, 1516821, 855913, 1517603, 66284, 1517085, 65709, 57670, 65382,
1043141, 65599, 1517678, 1750398, 952585, 1683169, 1517567, 855827, 1249556,
1457266, 952653, 1316925, 1749771, 66320, 65917, 65623, 1457176, 65344, 952643,
1682943, 1067538, 57559, 1749997, 355950, 1749668, 855606, 1749910, 1249481,
411965, 1517752, 1069571, 1749914, 1067563, 1069513, 1249327, 855888, 1683235,
1516521, 57328, 1043064, 1682888, 1015486, 411958, 1517382, 1516731, 1069621,
1249579, 1316785, 57519, 1683198, 57472, 855925, 1683136, 1457118, 855616,
855511, 855594, 57290, 1316822, 855847, 1249446, 1316486, 57335, 855445, 411964,
1015531, 1316673, 1683138, 855771, 1683249, 855413, 855753, 855711, 411981,
411939, 855686, 1015401, 1683154, 1249284, 1682954, 1683225, 1249513, 1015420,
1682840, 1682915, 1249447, 1457175, 855880, 1249333, 1249521, 855891, 855454,
1683163, 1457142, 1457135, 1682950, 1249509, 412008, 1015432, 1015389, 1457210,
1249354, 1457177, 1457148, 1249388, 1249565, 1249582, 1249342, 1457256, 1249476,
1249557, 411928, 1015460, 1457122, 1015475, 411978, 1015473, 411947, 1015499,
411950, 1457200, 1015436, 1457252, 1015496, 411946, 1457197, 411976, 1457269,
1457169, 1015430, 411986, 1015448, 1015484, 1015532, 1015449, 1015402, 1457240,
1457123, 1457182, 1457250, 411994, 1015451, 1015521, 412005, 411974, 411961,
411933, 411932, 411962, 411971, 411944]
                              ids  year  \
436266   allgemeinemissio0028unse  1874
463624   americancatholic0038vari  1927
1749883    amielsjournalbei00amie  1928
1496059          adambede0000elio  1887
589264   orationsfromhome0021mayo  1902
…                             …   …
411933     alexandertragdie00lang  1934
411932     alexandertragdie00lang  1934
411962     alexandertragdie00lang  1934
411971     alexandertragdie00lang  1934
411944     alexandertragdie00lang  1934

                                                      text
436266   Ich habe es nur mitgeteilt, um zu zeigen, was …
463624   And let each of us do his part to make our dea…
1749883  Here, then, is the service which Christi…
1496059  "I think your demand is just, Adam," sa…
589264   And yet there it stands, and nothing that Mr. …
…                                                   …
```

```
411933   Und was dann? \nIch fage Dir, feit geftern Abe…
411932   * Klebt Dir im Bart? \nNearchos Wir nennen's T…
411962   Das fchüttelt Dich ein wenig - dann ift's gut …
411971   Hephaiftion zögert deutlicher) \nDer Sprecher …
411944   Was! Was! \nHat ein liftiger Gott \nDich in fe…

[2090 rows x 3 columns]
2090
```

[14]:
```
# print(subset.year)
# for year in range(GLOBAL_MIN_YEAR, GLOBAL_MAX_YEAR+1):
#   print(year, counts.get(year, 0))
print(np.mean(subset.year))
print(np.std(subset.year))
plt.hist(subset.year, bins=GLOBAL_YEAR_RANGE+1)
plt.show()
```

```
1897.042105263158
27.497718867867714
```




[15]:
```
# Build a dataframe to back a dataset
class df_dataset(Dataset):
```

```
  def __init__(self, paradf):
    self.df = paradf

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

  def __getitem__(self, idx):
    return self.df.iloc[idx]
```

[16]:
```
# used_para = para_df[:1_000]
used_para = subset
train_stop = int(len(used_para) * 0.8)
val_stop = int(len(used_para) * 0.1)
test_stop = int(len(used_para) * 0.1)
train_books = used_para[:train_stop]
val_books = used_para[train_stop:(train_stop+val_stop)]
test_books = used_para[(train_stop+val_stop):]

train_dataset = df_dataset(train_books)
val_dataset = df_dataset(val_books)
test_dataset = df_dataset(test_books)
print(len(train_dataset))
print(len(val_dataset))
print(len(test_dataset))
```

```
1672
209
209
```

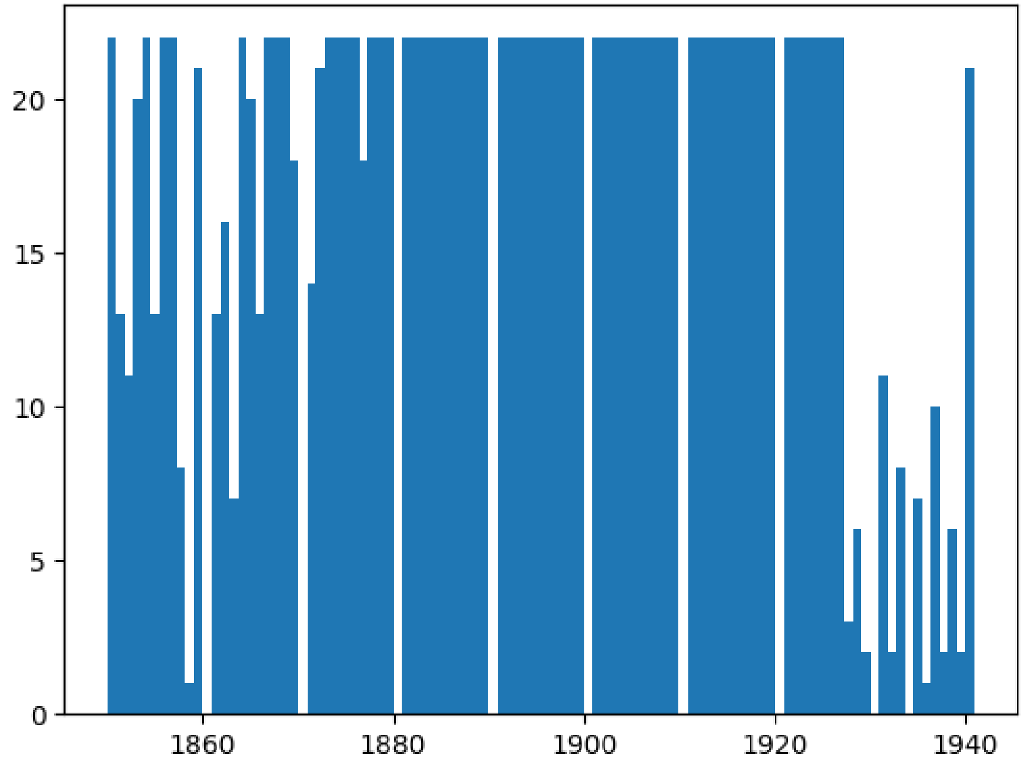
### 2.1 Check data distribution on training data

**NOTE:** Our data is super skewed

**TODO:** Instead of blindly filtering out out books by the first 1,000 should probably iteratively build a collection with an even distribution.

[17]:
```
print(np.mean(train_books.year))
print(np.std(train_books.year))
plt.hist(train_books['year'], bins=GLOBAL_YEAR_RANGE+1)
plt.show()
```

```
1892.8791866028707
23.412153783728026
```




## 3 Helper Functions

- Collate functions. A few different options:
    - Targets as vectors with normal distribution around correct year
    - Targets as one-hot vectors (where correct year is a 1)
    - Targets as single float value (ex: 1920.0)
- Main Functions:
    - Train
    - Evaluate Loss
    - Evaluate Accuracy

[18]:
```
# Define collate func

def discrete_collate(batch, tokenizer, std_dev=5):
  '''
  Targets as vectors with normal distribution around the correct year.
  Size of vector corresponds to the range of years in the full data.
  '''
  texts, labels, masks = [], [], []
  for data in batch:
    tokenizer_output = tokenizer([data['text']], truncation=True)
    tokenized_para = tokenizer_output['input_ids'][0]
```

```python
    texts.append(torch.tensor(tokenized_para))

    masks.append(torch.tensor(tokenizer_output['attention_mask'][0]))

    # Convert integer years to a normal distribution that centers around the␣
↪year
    skewed = (torch.tensor(data['year']) - GLOBAL_MIN_YEAR)
    normal = torch.distributions.normal.Normal(skewed, std_dev)
    values = torch.exp(normal.log_prob(torch.linspace(0, GLOBAL_YEAR_RANGE-1,␣
↪GLOBAL_YEAR_RANGE)))
    labels.append(values)

    # print(f'\nOriginal: {data["year"]}')
    # print(f'skewed: {skewed}')
    # print(f'center: {torch.argmax(values)}')

    # X = torch.arange(GLOBAL_YEAR_RANGE)
    # plt.scatter(X, values)
    # plt.scatter(skewed, values[skewed], color='red')
    # plt.axvline(skewed)
    # plt.show()

  texts = pad_sequence(texts, batch_first=True).to(device)
  labels = torch.stack(labels, dim=0).to(device)
  masks = pad_sequence(masks, batch_first=True, padding_value=0.0).to(device)

  return texts, labels, masks

def discrete_one_hot_collate(batch, tokenizer):
  '''
  Collate and make target labels as a one-hot vector.
  Values will be 0 for all years, and 1 for year of text.
  '''
  # NOTE: This might be worth using to show some example data for how much␣
↪worse this is than non-binary targets
  texts, labels, masks = [], [], []
  for data in batch:
    tokenizer_output = tokenizer(data['text'], truncation=True)
    tokenized_para = tokenizer_output['input_ids'][0]
    texts.append(torch.tensor(tokenized_para))

    mask = tokenizer_output['attention_mask'][0]
    masks.append(torch.tensor(mask))

    skewed = (torch.tensor(data['year'])- GLOBAL_MIN_YEAR)

    one_hot = F.one_hot(skewed, num_classes=GLOBAL_YEAR_RANGE).float()
```

```
    year_tens = torch.tensor(data['year']) - GLOBAL_MIN_YEAR
    one_hot = F.one_hot(year_tens, num_classes=GLOBAL_YEAR_RANGE).float()
    labels.append(one_hot)

  texts = pad_sequence(texts, batch_first=True).to(device)
  labels = torch.stack(labels, dim=0).to(device)
  masks = pad_sequence(masks, batch_first=True, padding_value=0.0).to(device)

  return texts, labels, masks


def continuous_collate(batch, tokenizer):
  texts, labels, masks = [], [], []
  real_years = []
  for data in batch:
    tokenizer_output = tokenizer([data['text']], truncation=True)
    tokenized_para = tokenizer_output['input_ids'][0]
    texts.append(torch.tensor(tokenized_para))

    mask = tokenizer_output['attention_mask'][0]
    masks.append(torch.tensor(mask))

    years = torch.tensor(data['year']).unsqueeze(0).float()
    normal_years = (years - GLOBAL_MIN_YEAR) / GLOBAL_YEAR_RANGE
    # normal_years = (years - GLOBAL_MIN_YEAR)
    labels.append(normal_years)
    real_years.append(years)

  texts = pad_sequence(texts, batch_first=True).to(device)
  labels = torch.stack(labels, dim=0).to(device)
  masks = pad_sequence(masks, batch_first=True, padding_value=0.0).to(device)
  real_years = torch.stack(real_years, dim=0).to(device)

  return texts, labels, masks, real_years
```

[19]:
```
def train(model, dataloader, optimizer, criterion, device, clip,␣
↪scheduler=None):
  model.train()
  epoch_loss = 0
  for batch in tqdm(dataloader):
    optimizer.zero_grad()

    texts, labels, masks = batch[0], batch[1], batch[2]
    output = model(texts.to(device), masks.to(device))

    loss = criterion(output, labels.to(device))
```

```
    lambda1 = 0.4
    lambda2 = 0.1
    # l1_params = torch.cat([x.view(-1) for x in model.linear_map.parameters()])
    # l1_reg = lambda1 * torch.norm(l1_params, p=1)
    # l2_reg = lambda2 * torch.norm(l1_params, p=2)
    # loss += l1_reg + l2_reg
    # loss += l1_reg

    # print(f'Got loss: {loss}')
    # print(f'output: {output}')
    # print(f'labels: {labels}')
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

    optimizer.step()
    if scheduler is not None:
      scheduler.step()

    epoch_loss += loss.item()

  return epoch_loss / len(dataloader)

def evaluate(model, dataloader, criterion, device):
  model.eval()

  epoch_loss = 0
  with torch.no_grad():
    for batch in tqdm(dataloader):
      texts, labels, masks = batch[0], batch[1], batch[2]

      output = model(texts.to(device), masks.to(device))
      loss = criterion(output, labels.to(device))
      lambda1 = 0.4
      lambda2 = 0.1
      # l1_params = torch.cat([x.view(-1) for x in model.linear_map.
↪parameters()])
      # l1_reg = lambda1 * torch.norm(l1_params, p=1)
      # l2_reg = lambda2 * torch.norm(l1_params, p=2)
      # loss += l1_reg + l2_reg
      # loss += l1_reg
      epoch_loss += loss.item()

  return epoch_loss / len(dataloader)

def evaluate_acc(model, dataloader, device):
  model.eval()
```

```python
  epoch_loss = 0
  with torch.no_grad():
    total_correct = 0
    total = 0
    for batch in tqdm(dataloader):
      texts, labels, masks = batch[0], batch[1], batch[2]
      labels_class = torch.argmax(labels, dim=1)

      output = model(texts.to(device), masks.to(device))
      output = F.softmax(output, dim=1)
      output_class = torch.argmax(output, dim=1)

      total_correct += torch.sum(torch.where(output_class == labels_class.
↪to(device), 1, 0))
      total += texts.size()[0]

  return total_correct / total

def evaluate_dist(model, dataloader, device):
  '''
  Evaluate based off of average distance from correct year
  '''
  model.eval()

  total_dist = 0
  total = 0
  with torch.no_grad():

    for batch in tqdm(dataloader):
      texts, labels, masks = batch
      correct = torch.argmax(labels, dim=1)

      output = model(texts.to(device), masks.to(device))
      output = F.softmax(output, dim=1)
      output_class = torch.argmax(output, dim=1)

      dists = torch.abs(correct - output_class)
      total_dist += dists.sum()
      total += dists.size()[0]

  return total_dist / total

def continuous_acc(model, dataloader, device):
  model.eval()

  epoch_loss = 0
  with torch.no_grad():
```

```
    total_correct = 0
    total = 0
    for batch in tqdm(dataloader):
      texts, labels, masks, real = batch

      output = model(texts.to(device), masks.to(device))
      # Rescale output back to specific years and count matches
      output = torch.round((output * GLOBAL_YEAR_RANGE) + GLOBAL_MIN_YEAR)

      counts = torch.where(output == real, 1, 0)
      total_correct += torch.sum(counts)
      total += len(output)

  return total_correct / total


def continuous_dist(model, dataloader, device):
  model.eval()

  total_dist = 0
  total = 0
  with torch.no_grad():
    for batch in tqdm(dataloader):
      texts, labels, masks, real = batch

      # Rescale output back to specific years and compare
      output = model(texts.to(device), masks.to(device))
      output = torch.round((output * GLOBAL_YEAR_RANGE) + GLOBAL_MIN_YEAR)
      dists = torch.abs(real - output)
      total_dist += dists.sum()
      total += dists.size()[0]

  return total_dist / total
```

[20]:
```
def init_weights(m: nn.Module, hidden_size=768):
    k = 1/hidden_size
    for name, param in m.named_parameters():
        if 'weight' in name:
            print(name)
            nn.init.uniform_(param.data, a=-1*k**0.5, b=k**0.5)
        else:
            print(name)
            nn.init.uniform_(param.data, 0)

def init_classification_head_weights(m: nn.Module, hidden_size=768):
    ### YOUR CODE STARTS HERE ###
    k = 1 / hidden_size
```

```
    for name, param in m.named_parameters():
        if 'classifier' in name:
            if 'weight' in name:
                print(name)
                nn.init.uniform_(param.data, a=-1*k**0.5, b=k**0.5)
            else:
                print(name)
                nn.init.uniform_(param.data, 0)

def count_params(model):
  return sum(p.numel() for p in model.parameters() if p.requires_grad)

def total_params(model):
  return sum(p.numel() for p in model.parameters())
```

# 4 Defining Model(s)

We should probably just use existing open source models (BERT, GPT-2?) for getting embedding's.

We will have to train our own model(s) to transform embeddings to year-label predictions. **NOTE:** I'm not sure what the most reasonable output for this would be. A full one-hot matrix for the full year range (~1850-2023)? Or maybe some continuous output that is mapped to a year? I would imagine that label probabilities would look close to a normal distribution (i.e. wouldn't expect a text to be 1920 OR 2020, probable just something like 1920-1923).

[21]:
```
from transformers import AutoModel, AutoTokenizer, IBertModel, AutoConfig
model_name = 'distilbert-base-uncased'
config = AutoConfig.from_pretrained(model_name, output_hidden_states=True)
distilbert_model = AutoModel.from_pretrained(model_name, config=config)
distilbert_continuous = AutoModel.from_pretrained(model_name, config=config)
distil_tokenizer = AutoTokenizer.from_pretrained(model_name)

ibert_model_name = 'kssteven/ibert-roberta-base'
ibert_config = AutoConfig.from_pretrained(ibert_model_name,␣
 ↪output_hidden_states=True)
ibert_tokenizer = AutoTokenizer.from_pretrained(ibert_model_name)
ibert_submodel = IBertModel.from_pretrained(ibert_model_name,␣
 ↪config=ibert_config)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.bias',
'vocab_transform.bias', 'vocab_projector.weight', 'vocab_layer_norm.weight',
'vocab_transform.weight', 'vocab_layer_norm.bias']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
```

```
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.bias',
'vocab_transform.bias', 'vocab_projector.weight', 'vocab_layer_norm.weight',
'vocab_transform.weight', 'vocab_layer_norm.bias']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
Some weights of the model checkpoint at kssteven/ibert-roberta-base were not
used when initializing IBertModel: ['lm_head.bias', 'lm_head.layer_norm.bias',
'lm_head.dense.weight', 'lm_head.decoder.weight', 'lm_head.layer_norm.weight',
'lm_head.dense.bias']
- This IS expected if you are initializing IBertModel from the checkpoint of a
model trained on another task or with another architecture (e.g. initializing a
BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing IBertModel from the checkpoint of
a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

[22]:
```python
# Define model
import torch.nn as nn
class DiscreteModel(nn.Module):
  def __init__(self, submodel, enc_hid_dim=768, dropout=0.1, flatten_all=False,␣
 ↪n_layers=7):
    super().__init__()

    self.flatten_all = flatten_all
    self.submodel = submodel
    self.enc_hid_dim = enc_hid_dim if not flatten_all else (enc_hid_dim *␣
 ↪n_layers)
    self.n_layers = n_layers

    for param in self.submodel.parameters():
      param.requires_grad = False

    self.dropout = nn.Dropout(dropout)
    # Single layer approach
    self.linear_map = nn.Linear(self.enc_hid_dim, GLOBAL_YEAR_RANGE)

    # Two layer approach
    # self.intermediate = nn.Linear(self.enc_hid_dim, GLOBAL_YEAR_RANGE*2)
```

```python
    # self.linear_map = nn.Linear(GLOBAL_YEAR_RANGE*2, GLOBAL_YEAR_RANGE)

    self.soft = nn.Softmax(dim=1)

  def forward(self, src, mask):
    submodel_out = self.submodel(src, mask, output_hidden_states=True)

    if self.flatten_all:
      sub_vect = torch.cat(submodel_out.hidden_states[-self.n_layers:], dim=2)[:
↪, 0]
    else:
      # With only last hidden state
      sub_vect = submodel_out.last_hidden_state[:, 0]

    sub_vect = self.dropout(sub_vect)

    # For two layer approach
    # sub_vect = self.intermediate(sub_vect)

    logits = self.linear_map(sub_vect)
    logits = self.soft(logits)

    return logits

class ContinuousModel(nn.Module):
  def __init__(self, submodel, enc_hid_dim=768, hid_dim=100, dropout=0.1):
    super().__init__()

    self.submodel = submodel
    self.enc_hid_dim = enc_hid_dim
    self.hid_dim = hid_dim

    for param in self.submodel.parameters():
      param.requires_grad = False

    self.dropout = nn.Dropout(dropout)

    # Two layer approach
    # self.intermediate = nn.Linear(enc_hid_dim, self.hid_dim)
    # self.linear_map = nn.Linear(self.hid_dim, 1)

    # Single layer approach
    self.linear_map = nn.Linear(enc_hid_dim, 1)

  def forward(self, src, mask):
    submodel_out = self.submodel(src, mask)
    sub_vect = submodel_out.last_hidden_state[:, 0]
```

```
    sub_vect = self.dropout(sub_vect)

    # For two layer approach
    # sub_vect = self.intermediate(sub_vect)

    pred = self.linear_map(sub_vect)

    return pred
```

[23]:

```
# Make instance of model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'USING DEVICE: {str(device).upper()}')
```

```
USING DEVICE: CUDA
```

# 5 Model Training

## 5.1 Distrete Classification Model

[41]:

```
# model = DiscreteModel(distilbert_model, dropout=0.1, flatten_all=True,␣
 ↪n_layers=3).to(device)
# model = DiscreteModel(distilbert_model, dropout=0.15, flatten_all=False).
 ↪to(device)
model = DiscreteModel(distilbert_model, dropout=0.2, flatten_all=False).
 ↪to(device)
model.apply(init_classification_head_weights)
model.to(device)
print(f'Total trainable params: {count_params(model):,}')
print(f'          Total params: {total_params(model):,}')
print('Model initialized\n')
```

```
Total trainable params: 76,131
          Total params: 66,439,011
Model initialized
```

[42]:

```
BATCH_SIZE = 10
LR = 1e-3
N_EPOCHS = 10
CLIP = 1.0
WEIGHT_DECAY = 0
```

[43]:

```
train_dataloader = DataLoader(
  train_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(discrete_collate, tokenizer=distil_tokenizer),
  shuffle=True
```

```
)
val_dataloader = DataLoader(
  val_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(discrete_collate, tokenizer=distil_tokenizer)
)
test_dataloader = DataLoader(
  test_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(discrete_collate, tokenizer=distil_tokenizer)
)

optimizer = optim.Adam(model.parameters(), lr=LR, weight_decay=WEIGHT_DECAY)
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=10,␣
 ↪num_training_steps=N_EPOCHS*len(train_dataloader))
class_criterion = nn.MSELoss()
# class_criterion = nn.CrossEntropyLoss()
```

[44]:

```
# Set model to fine-tune distil-bert
# for param in model.parameters():
#   param.requires_grad = True

print(f'The model has {count_params(model):,} trainable parameters')
print(f'Total num of params: {total_params(model):,}')
print(model.enc_hid_dim)
print(model)
```

```
The model has 76,131 trainable parameters
Total num of params: 66,439,011
768
DiscreteModel(
  (submodel): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
```

```
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
      )
    )
  )
  (dropout): Dropout(p=0.2, inplace=False)
  (linear_map): Linear(in_features=768, out_features=99, bias=True)
  (soft): Softmax(dim=1)
)
```

[45]:
```
train_loss = evaluate(model, train_dataloader, class_criterion, device)
train_error = evaluate_dist(model, train_dataloader, device)

valid_loss = evaluate(model, val_dataloader, class_criterion, device)
valid_error = evaluate_dist(model, val_dataloader, device)

print(f'Initial Train Loss: {train_loss:.5f}')
print(f'Initial Train Error Margin (years): {train_error:.3f}')
print(f'Initial Valid Loss: {valid_loss:.5f}')
print(f'Initial Valid Error Margin (years): {valid_error:.3f}')

train_losses = []
train_errors = []
valid_losses = []
valid_errors = []
for epoch in range(N_EPOCHS):

  # Training
  start_time = time.time()
  train_loss = train(model, train_dataloader, optimizer, class_criterion,␣
 ↪device, CLIP, scheduler)
  # train_loss = train(model, train_dataloader, optimizer, class_criterion,␣
 ↪device, CLIP)
  end_time = time.time()
  train_error = evaluate_dist(model, train_dataloader, device)

  # Validation
  valid_loss = evaluate(model, val_dataloader, class_criterion, device)
```

```python
    valid_error = evaluate_dist(model, val_dataloader, device)

    # Track values
    train_losses.append(train_loss)
    train_errors.append(train_error.item())
    valid_losses.append(valid_loss)
    valid_errors.append(valid_error.item())

    # Print info
    epoch_secs = end_time - start_time
    print(f'Epoch: {epoch+1:02} / {N_EPOCHS} | Time: {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.5f}')
    print(f'\tTrain Error Margin: {train_error:.3f}')
    print(f'\tValid Loss: {valid_loss:.5f}')
    print(f'\tValid Error Margin: {valid_error:.3f}')

test_loss = evaluate(model, test_dataloader, class_criterion, device)
test_error = evaluate_dist(model, test_dataloader, device)
print(f'TESTING:')
print(f'\t    Testing Loss: {test_loss:.5f}')
print(f'\tTesting Error Margin: {test_error:.3f}')
```

```
  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Initial Train Loss: 0.00047
Initial Train Error Margin (years): 26.080
Initial Valid Loss: 0.00048
Initial Valid Error Margin (years): 37.789

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 01 / 10 | Time: 21.10928249359131s
        Train Loss: 0.00044
        Train Error Margin: 16.650
        Valid Loss: 0.00050
        Valid Error Margin: 26.263

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]
```

```
  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 02 / 10 | Time: 20.75991702079773s
        Train Loss: 0.00042
        Train Error Margin: 16.186
        Valid Loss: 0.00052
        Valid Error Margin: 27.053

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 03 / 10 | Time: 21.04240322113037s
        Train Loss: 0.00041
        Train Error Margin: 16.146
        Valid Loss: 0.00049
        Valid Error Margin: 26.636

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 04 / 10 | Time: 20.791017055511475s
        Train Loss: 0.00041
        Train Error Margin: 15.994
        Valid Loss: 0.00048
        Valid Error Margin: 23.766

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 05 / 10 | Time: 20.745238780975342s
        Train Loss: 0.00040
        Train Error Margin: 15.990
        Valid Loss: 0.00047
        Valid Error Margin: 22.187

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]
```

```
  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 06 / 10 | Time: 20.75317621231079s
        Train Loss: 0.00040
        Train Error Margin: 15.267
        Valid Loss: 0.00047
        Valid Error Margin: 23.297

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 07 / 10 | Time: 20.62689518928528s
        Train Loss: 0.00039
        Train Error Margin: 15.217
        Valid Loss: 0.00048
        Valid Error Margin: 22.789

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 08 / 10 | Time: 20.302544832229614s
        Train Loss: 0.00039
        Train Error Margin: 15.395
        Valid Loss: 0.00047
        Valid Error Margin: 22.062

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 09 / 10 | Time: 20.57172155380249s
        Train Loss: 0.00039
        Train Error Margin: 15.008
        Valid Loss: 0.00048
        Valid Error Margin: 23.569

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]
```

```
Epoch: 10 / 10 | Time: 20.8786883354187s
        Train Loss: 0.00039
        Train Error Margin: 14.989
        Valid Loss: 0.00048
        Valid Error Margin: 23.349
```

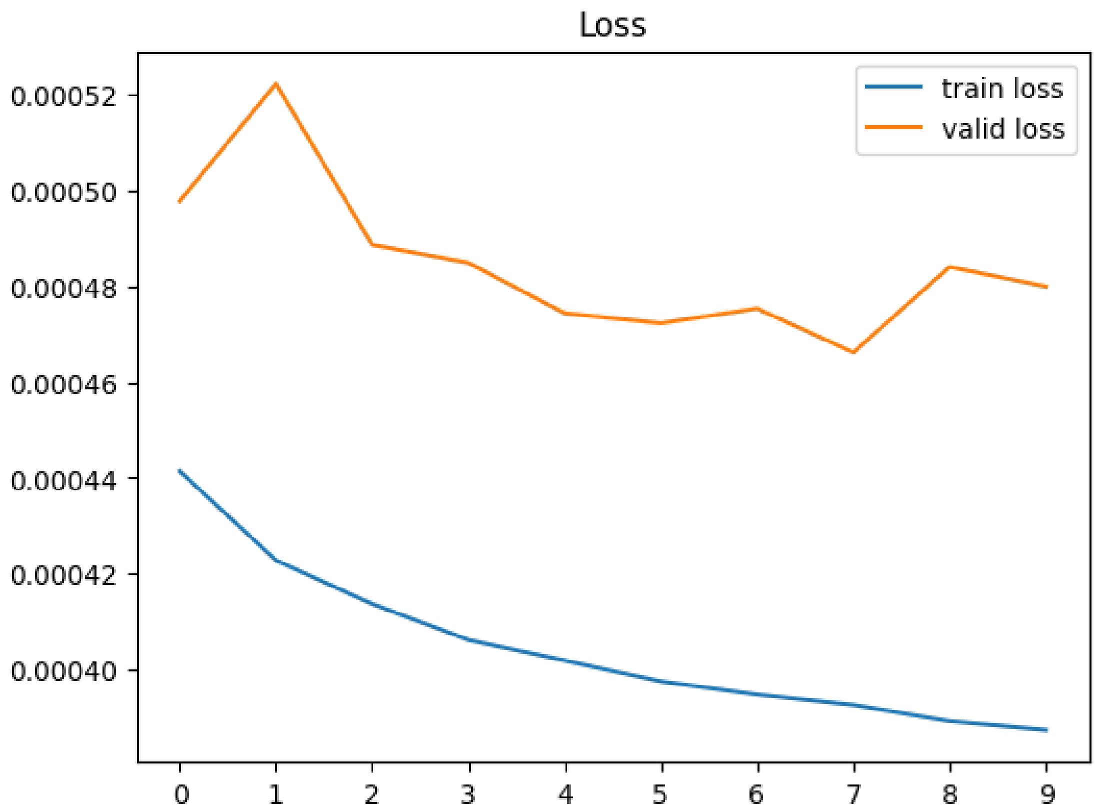
```
  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]
```

```
TESTING:
            Testing Loss: 0.00053
        Testing Error Margin: 32.684
```

[46]:

```python
print(f'train loss: {train_losses}')
print(f'valid loss: {valid_losses}')
plt.plot(train_losses, label='train loss')
plt.plot(valid_losses, label='valid loss')
plt.xticks(range(len(train_losses)))
plt.legend()
plt.title('Loss')
plt.show()
```

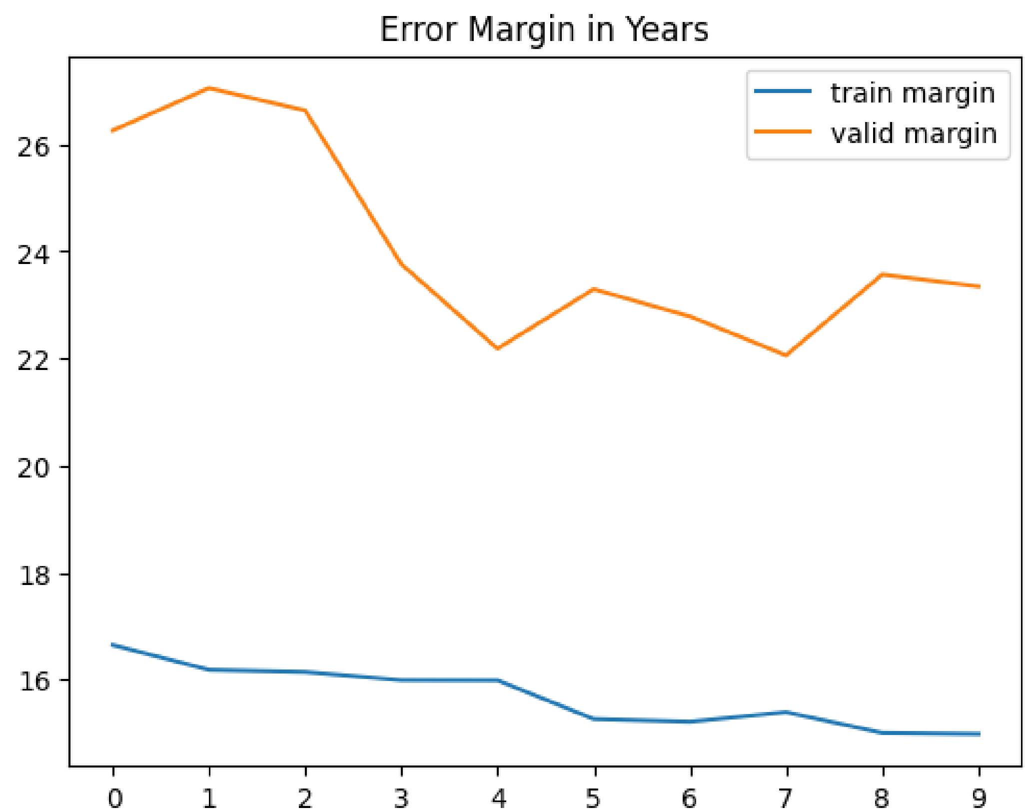
```
train loss: [0.0004414083249728373, 0.0004228181756583841,
0.0004137140191839232, 0.000406209964109751, 0.0004019205497613 82,
0.0003975484564545 6084, 0.0003948355510406 7946, 0.0003926490722473578,
0.00038928926109552516, 0.00038744427919958806]
valid loss: [0.0004977321257770416, 0.0005222353003253895,
0.0004885924988359745, 0.00048485320543737283, 0.0004743307709716083,
0.00047229666129818985, 0.0004752982044703371, 0.0004661811662593945,
0.00048401698725120655, 0.00047989686468749175]
```




[47]:

```
print(f'train error margin: {train_errors}')
print(f'valid error margin: {valid_errors}')
plt.plot(train_errors, label='train margin')
plt.plot(valid_errors, label='valid margin')
plt.xticks(range(len(train_errors)))
plt.legend()
plt.title('Error Margin in Years')
plt.show()
```

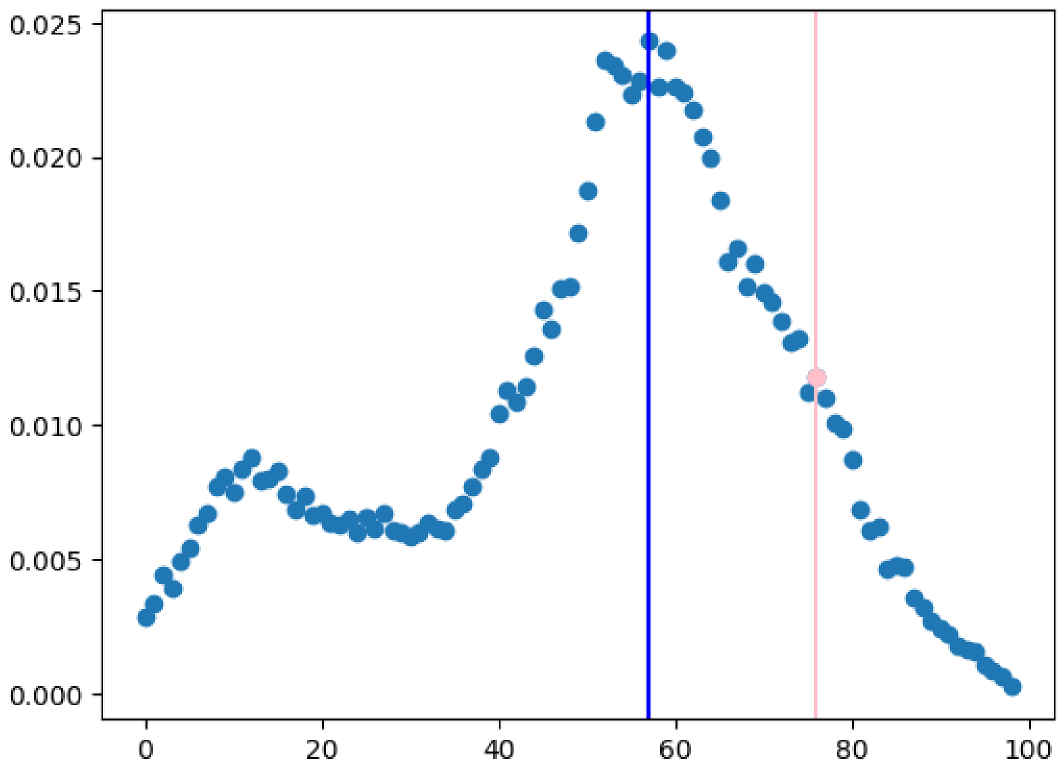
```
train error margin: [16.649520874023438, 16.186004638671875, 16.145933151245117,
15.9940185546875, 15.990429878234863, 15.267343521118164, 15.216506958007812,
15.394736289978027, 15.00837230682373, 14.988636016845703]
valid error margin: [26.26315689086914, 27.052631378173828, 26.636362075805664,
23.765548706054688, 22.186601638793945, 23.296649932861328, 22.789472579956055,
22.06220054626465, 23.569377899169922, 23.349281311035156]
```



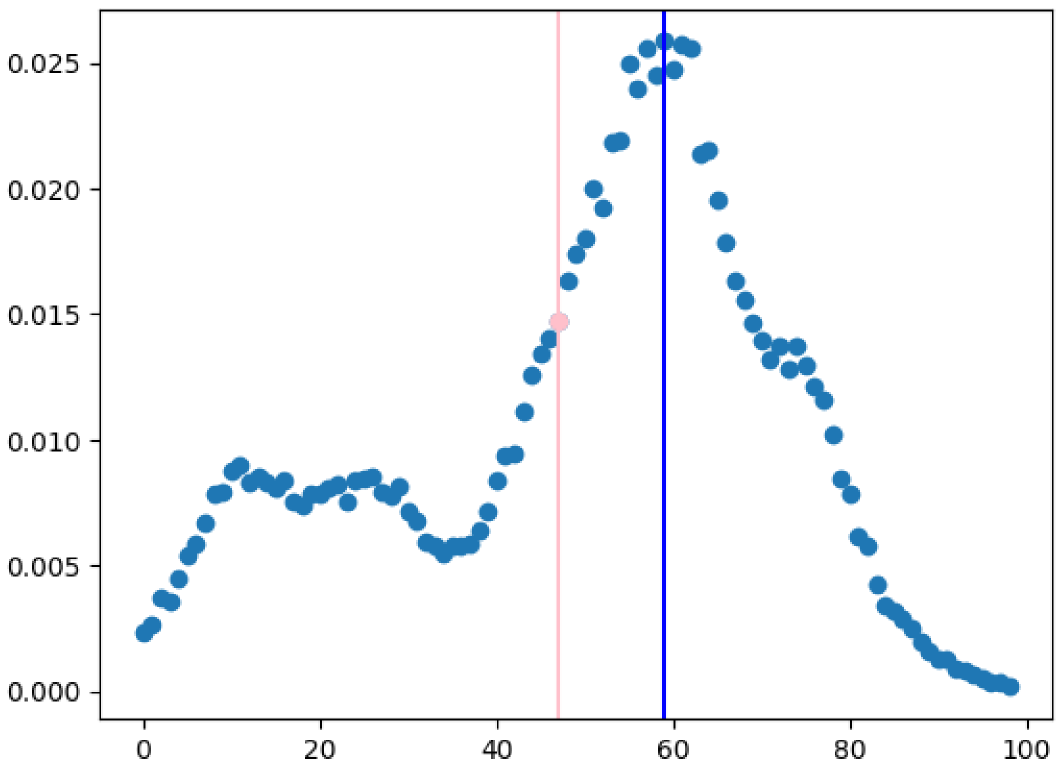

[48]:
```
# Just get one random batch from the dataloader
# for item in test_dataloader:
for item in train_dataloader:
  # print(item)
  text, year, mask = item
  break

pred = model(text, mask)
# print(year)
# print(pred)
# print(year.shape)
# print(pred.shape)

for inputs, values in zip(year, pred):
  correct = torch.argmax(inputs).item()
  correct_guess_val = values.tolist()[correct]
  guess = torch.argmax(values).item()
  guess_val = values.tolist()[guess]
  print(f'correct: {correct}')
  print(f'argmax: {guess}')
```

```python
X = torch.arange(GLOBAL_YEAR_RANGE).tolist()
plt.scatter(X, values.tolist())
plt.axvline(x=guess, color='blue')
plt.axvline(x=correct, color='pink')
plt.scatter(correct, correct_guess_val, color='pink')
# plt.scatter(X, inputs.tolist(), color='orange')
plt.show()
```

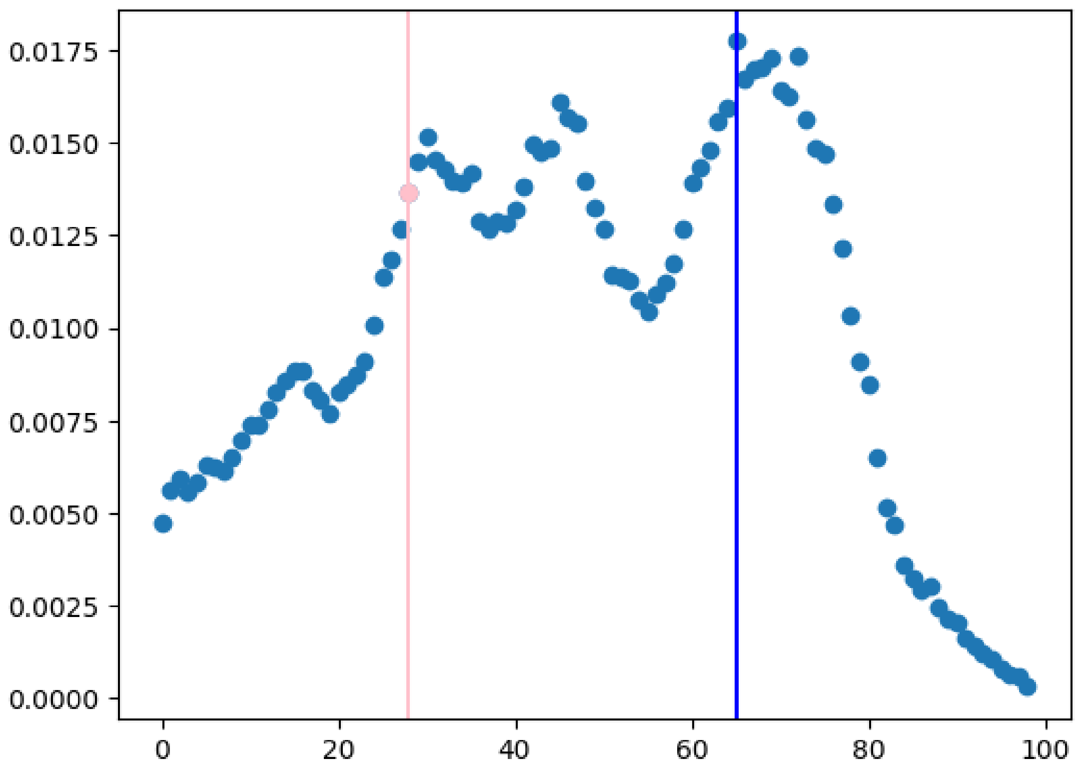
```
correct: 76
argmax: 57
```



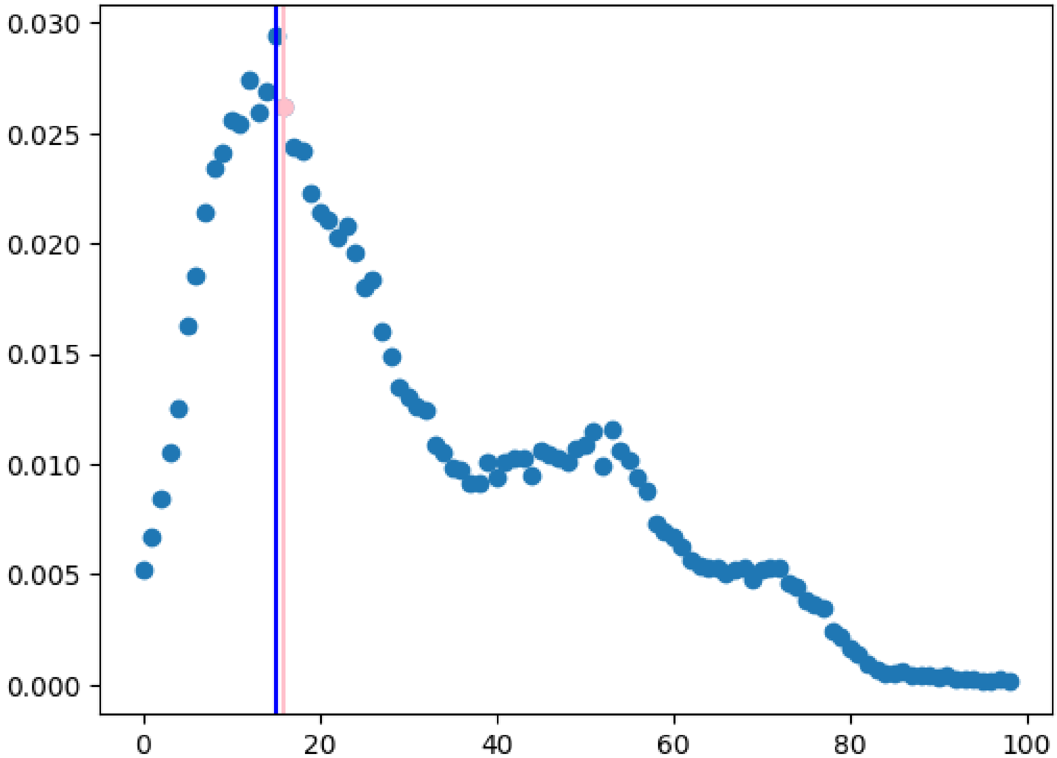

```
correct: 47
argmax: 59
```



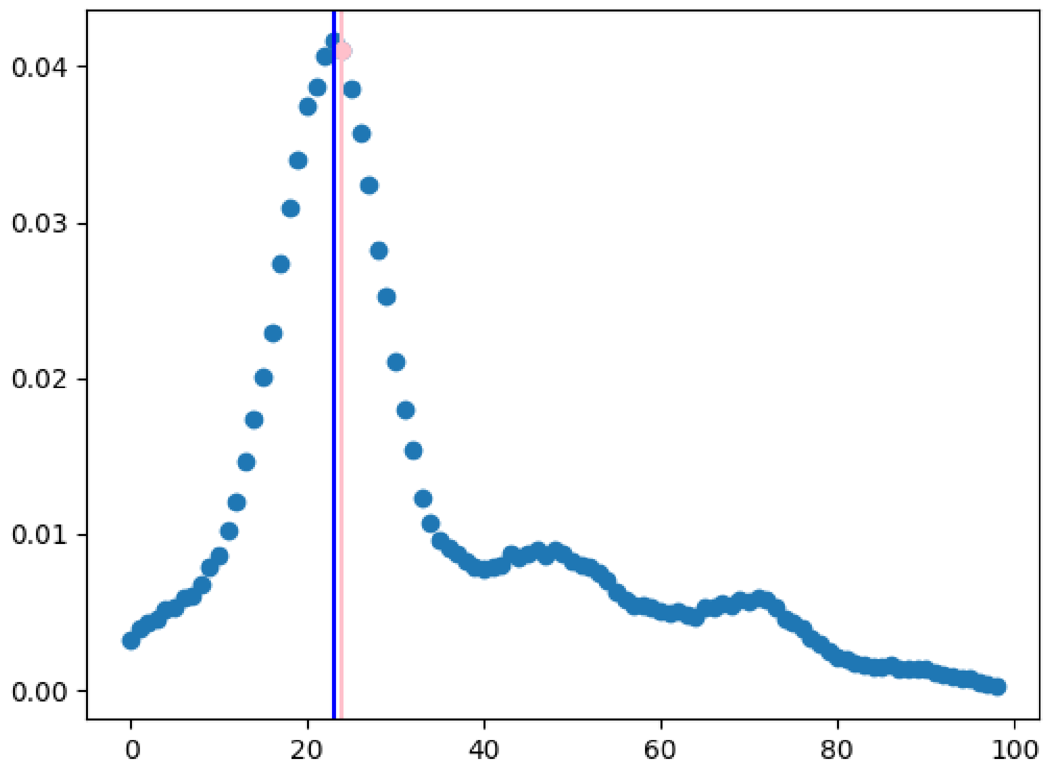

```
correct: 28
argmax: 65
```



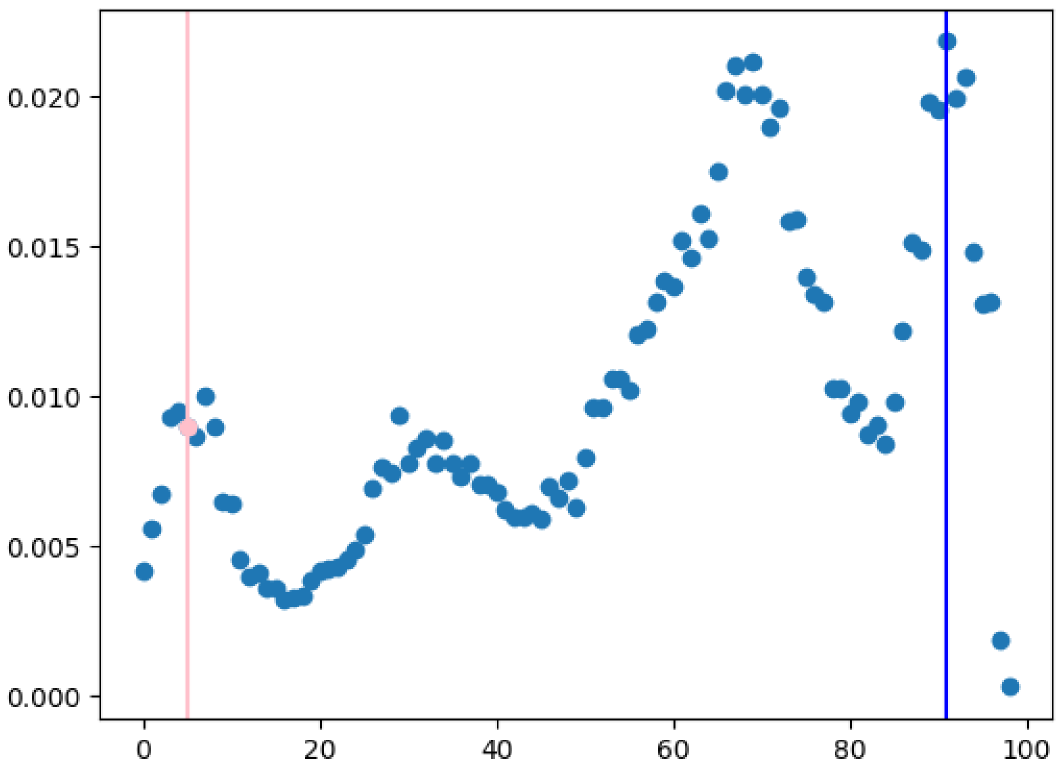

```
correct: 16
argmax: 15
```



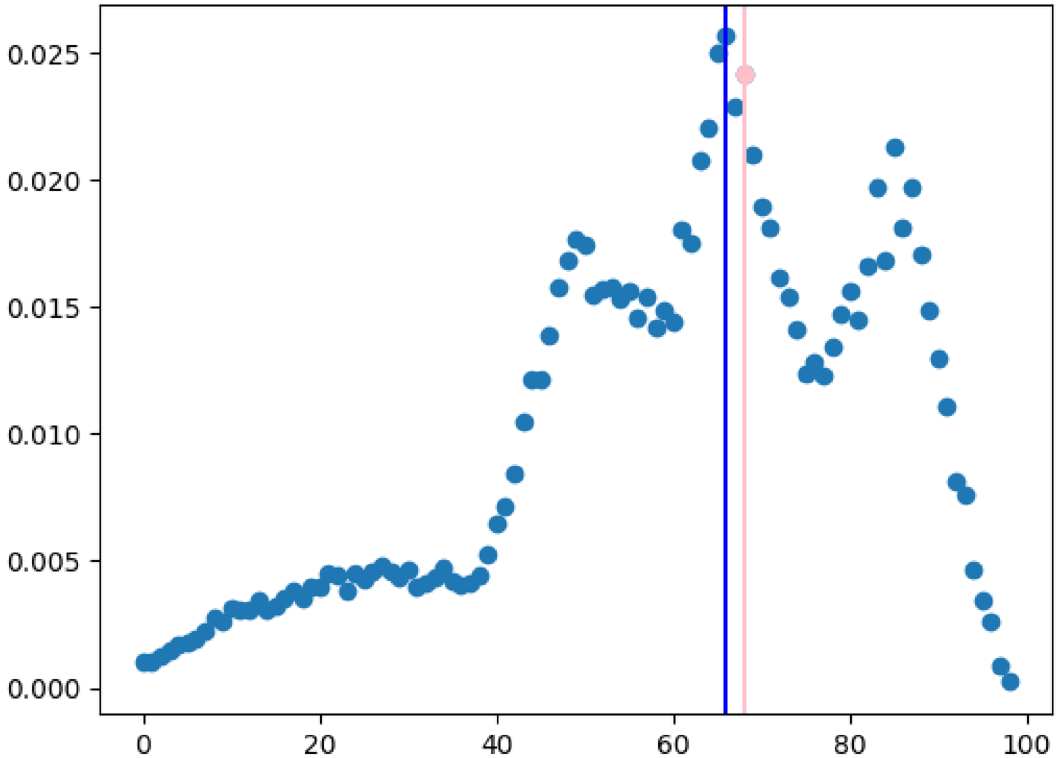

```
correct: 24
argmax: 23
```



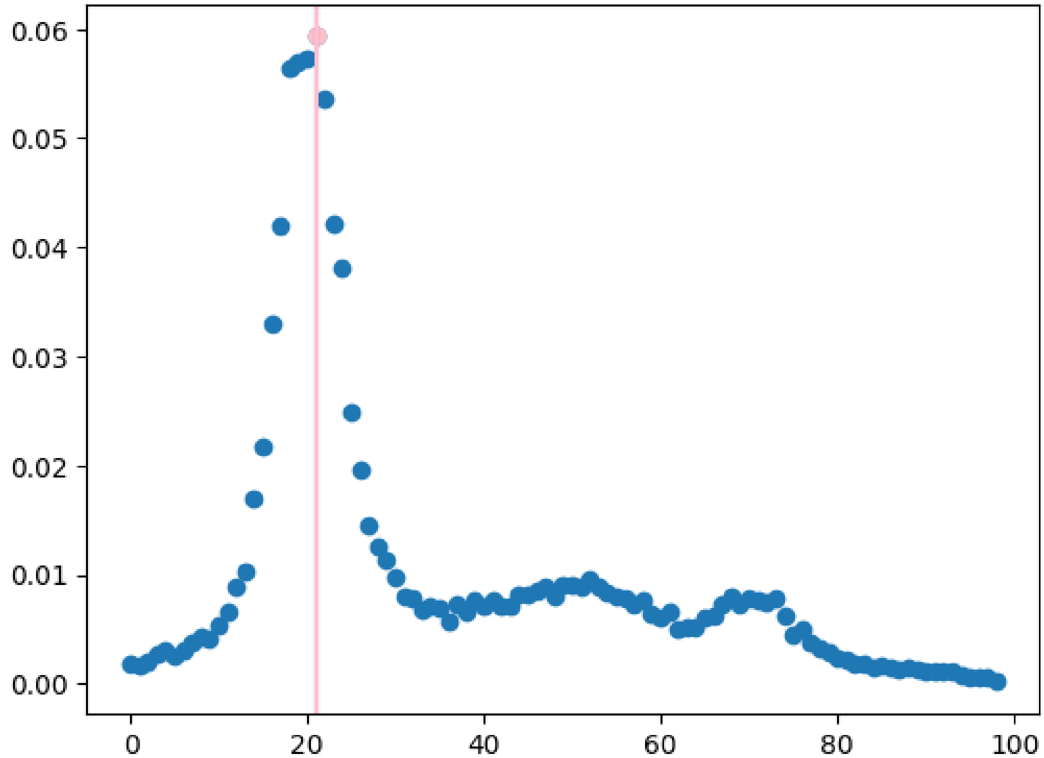

```
correct: 5
argmax: 91
```



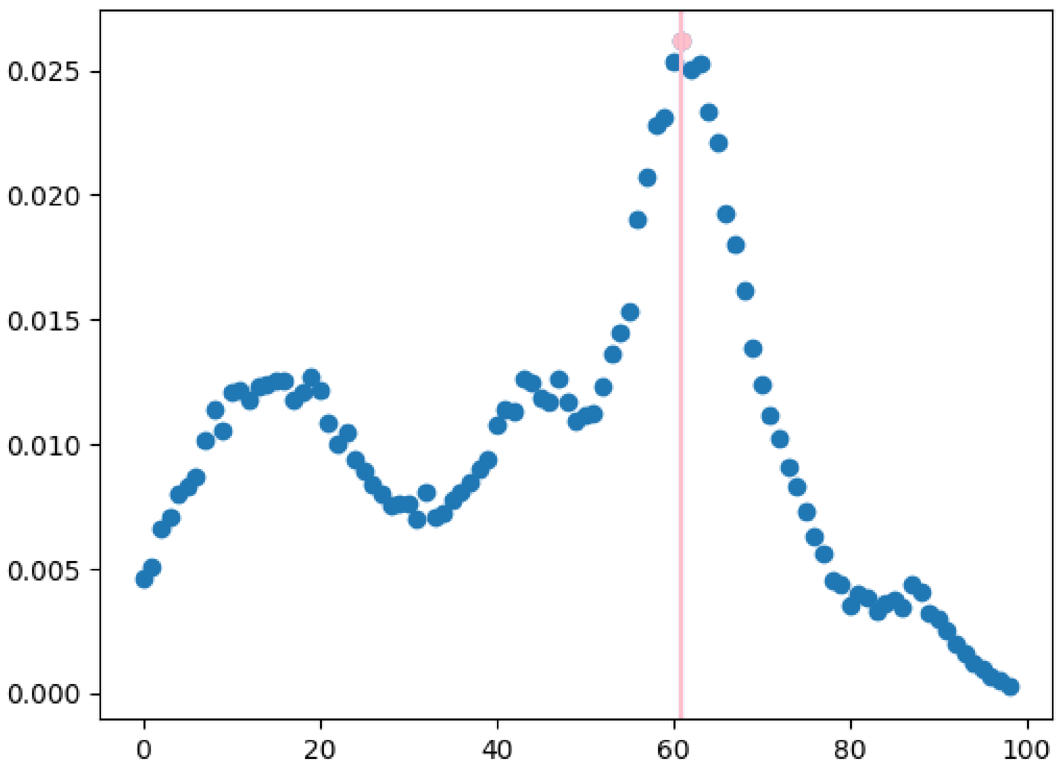

```
correct: 68
argmax: 66
```



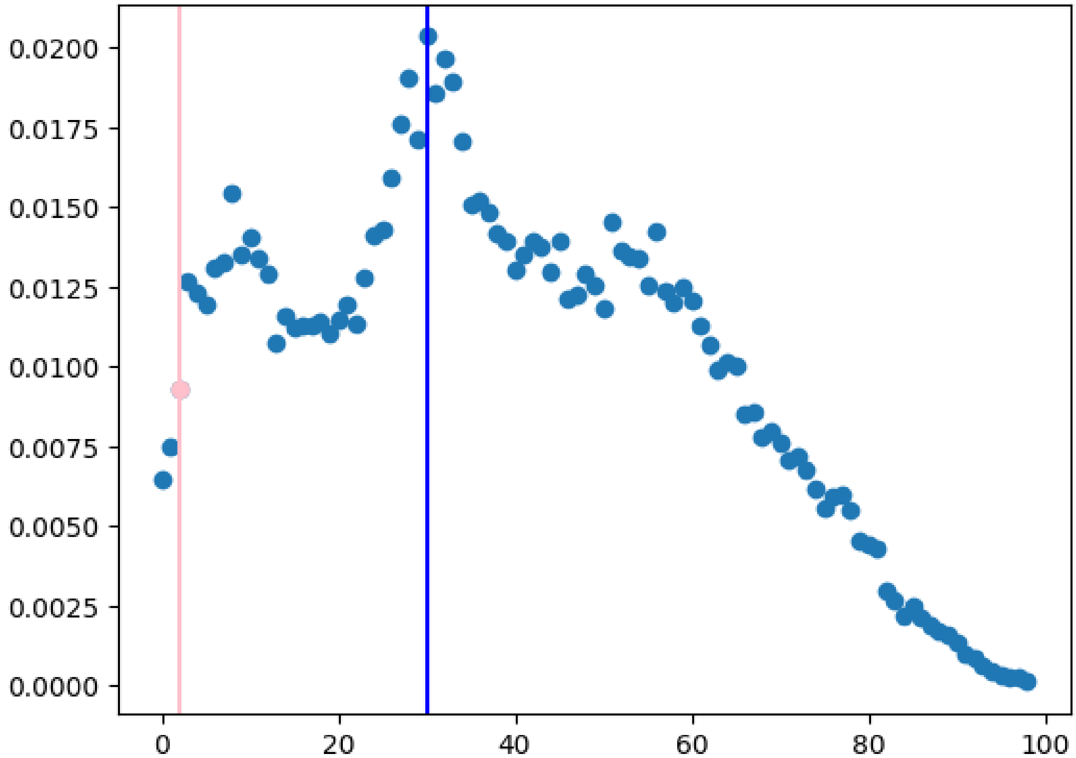

```
correct: 21
argmax: 21
```




```
correct: 61
argmax: 61
```




```
correct: 2
argmax: 30
```




### 5.2 Continuous Prediction Model

[49]:
```
continuous_model = ContinuousModel(distilbert_continuous).to(device)
continuous_model.apply(init_classification_head_weights)
continuous_model.to(device)
print(f'Total trainable params: {count_params(continuous_model):,}')
print(f'          Total params: {total_params(continuous_model):,}')
print('Continuous model initialized\n')
```

```
Total trainable params: 769
          Total params: 66,363,649
Continuous model initialized
```

[50]:
```
BATCH_SIZE = 16
LR = 1e-3
N_EPOCHS = 25
CLIP = 1.0
```

[51]:
```
train_dataloader = DataLoader(
  train_dataset,
  batch_size=BATCH_SIZE,
```

```
  collate_fn=partial(continuous_collate, tokenizer=distil_tokenizer),
  shuffle=False
)
val_dataloader = DataLoader(
  val_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(continuous_collate, tokenizer=distil_tokenizer)
)
test_dataloader = DataLoader(
  test_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(continuous_collate, tokenizer=distil_tokenizer)
)

optimizer = optim.Adam(continuous_model.parameters(), lr=LR)
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=10,␣
 ↪num_training_steps=N_EPOCHS*len(train_dataloader))
continuous_criterion = torch.nn.MSELoss()
```

[52]:
```
print(f'The model has {count_params(continuous_model):,} trainable parameters')
print(f'Total num of params: {total_params(continuous_model):,}')
```

```
The model has 769 trainable parameters
Total num of params: 66,363,649
```

[53]:
```
train_loss = evaluate(continuous_model, train_dataloader, continuous_criterion,␣
 ↪device)
train_error = continuous_dist(continuous_model, train_dataloader, device)
# train_acc = continuous_acc(continuous_model, train_dataloader, device)

valid_loss = evaluate(continuous_model, val_dataloader, continuous_criterion,␣
 ↪device)
valid_error = continuous_dist(continuous_model, val_dataloader, device)
# valid_acc = continuous_acc(continuous_model, val_dataloader, device)

print(f'Initial Train Loss: {train_loss:.5f}')
print(f'Initial Train Error Margin (years): {train_error:.3f}')
# print(f'Initial Train Acc: {train_acc:.3f}')
print(f'Initial Valid Loss: {valid_loss:.5f}')
print(f'Initial Valid Error Margin (years): {valid_error:.3f}')
# print(f'Initial Valid Acc: {valid_acc:.3f}')

train_losses = []
train_accs = []
train_errors = []
valid_losses = []
valid_accs = []
```

```
valid_errors = []
for epoch in range(N_EPOCHS):
  start_time = time.time()
  train_loss = train(continuous_model, train_dataloader, optimizer,
 ↪continuous_criterion, device, CLIP, scheduler)
  end_time = time.time()
  train_error = continuous_dist(continuous_model, train_dataloader, device)
  # train_acc = continuous_acc(continuous_model, train_dataloader, device)

  valid_loss = evaluate(continuous_model, val_dataloader, continuous_criterion,
 ↪device)
  valid_error = continuous_dist(continuous_model, val_dataloader, device)
  # valid_acc = continuous_acc(continuous_model, val_dataloader, device)

  train_losses.append(train_loss)
  train_errors.append(train_error.item())
  # train_accs.append(train_acc.item())
  valid_losses.append(valid_loss)
  valid_errors.append(valid_error.item())
  # valid_accs.append(valid_acc.item())

  epoch_secs = end_time - start_time

  print(f'Epoch: {epoch+1:02}  / {N_EPOCHS}| Time: {epoch_secs}s')
  print(f'\tTrain Loss: {train_loss:.5f}')
  print(f'\tTrain Error Margin: {train_error:.3f}')
  # print(f'\tTrain Acc: {train_acc:.3f}')
  print(f'\tValid Loss: {valid_loss:.5f}')
  print(f'\tValid Error Margin: {valid_error:.3f}')
  # print(f'\tValid Acc: {valid_acc:.3f}')

test_loss = evaluate(continuous_model, test_dataloader, continuous_criterion,
 ↪device)
# test_acc = continuous_acc(continuous_model, test_dataloader, device)
test_error = continuous_dist(continuous_model, test_dataloader, device).item()
print(f'TESTING:')
print(f'\t      Testing Loss: {test_loss:.5f}')
# print(f'\t  Testing Accuracy: {test_acc:.3f}')
print(f'\tTesting Err Margin: {test_error:.3f}')
```

```
  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Initial Train Loss: 0.17894
```

```
Initial Train Error Margin (years): 35.399
Initial Valid Loss: 0.28564
Initial Valid Error Margin (years): 39.344
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 01  / 25| Time: 22.655990600585938s
        Train Loss: 0.06677
        Train Error Margin: 21.824
        Valid Loss: 0.10016
        Valid Error Margin: 24.943
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 02  / 25| Time: 23.16628336906433s
        Train Loss: 0.05034
        Train Error Margin: 22.339
        Valid Loss: 0.09975
        Valid Error Margin: 24.756
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 03  / 25| Time: 23.024039268493652s
        Train Loss: 0.04797
        Train Error Margin: 21.035
        Valid Loss: 0.09478
        Valid Error Margin: 24.249
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 04  / 25| Time: 23.049909830093384s
        Train Loss: 0.04630
        Train Error Margin: 22.901
```

```
        Valid Loss: 0.10244
        Valid Error Margin: 25.325

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 05  / 25| Time: 23.001198053359985s
        Train Loss: 0.04594
        Train Error Margin: 21.910
        Valid Loss: 0.09854
        Valid Error Margin: 24.809

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 06  / 25| Time: 22.95559549331665s
        Train Loss: 0.04484
        Train Error Margin: 19.178
        Valid Loss: 0.08943
        Valid Error Margin: 23.847

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 07  / 25| Time: 22.969202518463135s
        Train Loss: 0.04214
        Train Error Margin: 18.599
        Valid Loss: 0.08311
        Valid Error Margin: 23.167

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 08  / 25| Time: 23.00796604156494s
        Train Loss: 0.04270
        Train Error Margin: 19.185
        Valid Loss: 0.08543
        Valid Error Margin: 23.282
```

```
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 09  / 25| Time: 23.030274629592896s
        Train Loss: 0.04307
        Train Error Margin: 19.002
        Valid Loss: 0.08735
        Valid Error Margin: 23.665
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 10  / 25| Time: 23.02630352973938s
        Train Loss: 0.04286
        Train Error Margin: 18.273
        Valid Loss: 0.08529
        Valid Error Margin: 23.694
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 11  / 25| Time: 23.081655979156494s
        Train Loss: 0.04201
        Train Error Margin: 18.151
        Valid Loss: 0.08242
        Valid Error Margin: 23.311
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/105 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
  0%|          | 0/14 [00:00<?, ?it/s]
Epoch: 12  / 25| Time: 22.992695093154907s
        Train Loss: 0.04216
        Train Error Margin: 16.696
        Valid Loss: 0.07691
        Valid Error Margin: 23.249
  0%|          | 0/105 [00:00<?, ?it/s]
```

```
  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 13  / 25| Time: 22.980769634246826s
        Train Loss: 0.04105
        Train Error Margin: 17.441
        Valid Loss: 0.08228
        Valid Error Margin: 23.641

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 14  / 25| Time: 23.0172221660614s
        Train Loss: 0.04154
        Train Error Margin: 16.085
        Valid Loss: 0.07793
        Valid Error Margin: 23.799

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 15  / 25| Time: 23.044827461242676s
        Train Loss: 0.03983
        Train Error Margin: 16.388
        Valid Loss: 0.07677
        Valid Error Margin: 23.402

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 16  / 25| Time: 23.04594612121582s
        Train Loss: 0.03874
        Train Error Margin: 16.728
        Valid Loss: 0.07830
        Valid Error Margin: 23.584

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]
```

```
  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 17  / 25| Time: 23.06488609313965s
        Train Loss: 0.03916
        Train Error Margin: 17.227
        Valid Loss: 0.08315
        Valid Error Margin: 24.019

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 18  / 25| Time: 23.045387983322144s
        Train Loss: 0.03986
        Train Error Margin: 15.984
        Valid Loss: 0.07758
        Valid Error Margin: 24.019

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 19  / 25| Time: 23.09258770942688s
        Train Loss: 0.03736
        Train Error Margin: 16.452
        Valid Loss: 0.08193
        Valid Error Margin: 24.469

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 20  / 25| Time: 23.098137378692627s
        Train Loss: 0.03893
        Train Error Margin: 16.266
        Valid Loss: 0.08105
        Valid Error Margin: 24.445

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]
```

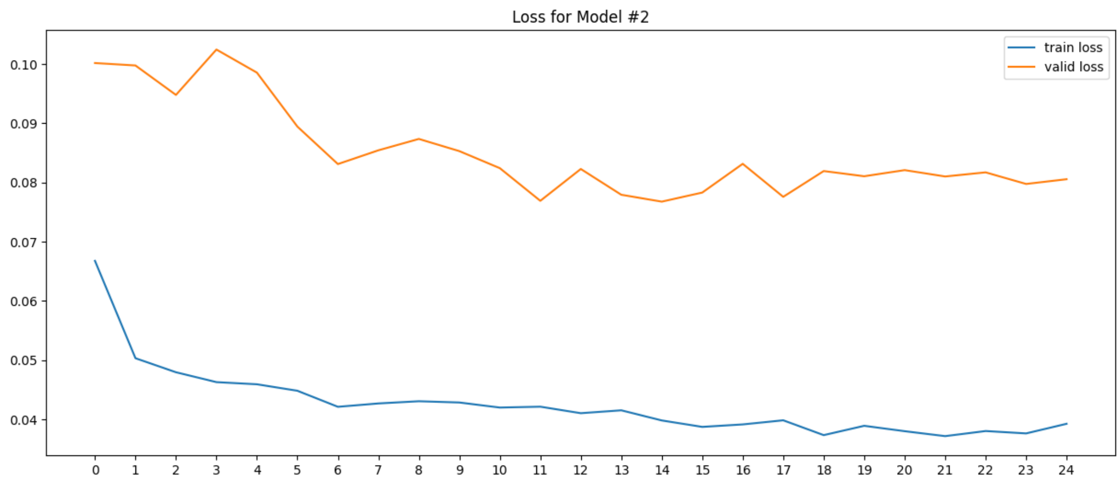
```
  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 21  / 25| Time: 23.074734926223755s
        Train Loss: 0.03802
        Train Error Margin: 16.129
        Valid Loss: 0.08209
        Valid Error Margin: 24.746

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 22  / 25| Time: 23.05326819419861s
        Train Loss: 0.03718
        Train Error Margin: 15.766
        Valid Loss: 0.08101
        Valid Error Margin: 24.866

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 23  / 25| Time: 23.060117483139038s
        Train Loss: 0.03805
        Train Error Margin: 15.648
        Valid Loss: 0.08171
        Valid Error Margin: 25.172

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

Epoch: 24  / 25| Time: 23.08887219429016s
        Train Loss: 0.03764
        Train Error Margin: 15.067
        Valid Loss: 0.07975
        Valid Error Margin: 25.440

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/105 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]
```

```
Epoch: 25  / 25| Time: 23.08445644378662s
        Train Loss: 0.03927
        Train Error Margin: 14.870
        Valid Loss: 0.08055
        Valid Error Margin: 26.258

  0%|          | 0/14 [00:00<?, ?it/s]

  0%|          | 0/14 [00:00<?, ?it/s]

TESTING:
              Testing Loss: 0.14615
        Testing Err Margin: 32.273
```

[57]:

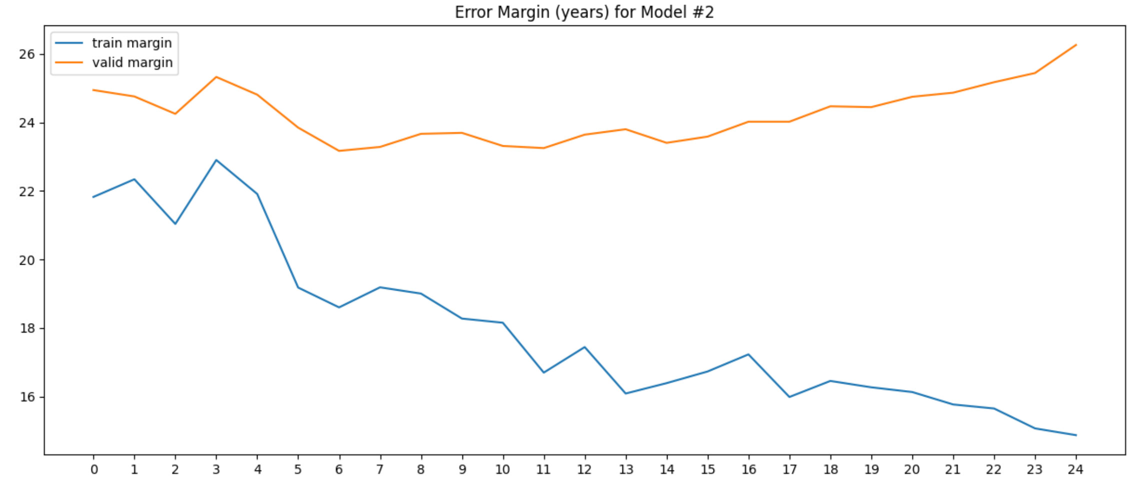
```
print(f'train loss: {train_losses}')
print(f'valid loss: {valid_losses}')
plt.figure(figsize=(15,6))
plt.plot(train_losses, label='train loss')
plt.plot(valid_losses, label='valid loss')
plt.xticks(range(len(train_losses)))
plt.title('Loss for Model #2')
plt.legend()
plt.show()
```

```
train loss: [0.06676834122765632, 0.05033716827276207, 0.04796875188393252,
0.04630095157772303, 0.045937419620652994, 0.04484200678943168,
0.04213663493948323, 0.04270101824686641, 0.04307161346077919,
0.04286059304362252, 0.04200892704760745, 0.042155050300061704,
0.04105477183170262, 0.0415429122568596, 0.039833315364306884,
0.03874388367292427, 0.039159937539980524, 0.03985804791251819,
0.03735535202459211, 0.038926915939719905, 0.03801896238610858,
0.03718469071955908, 0.03804616282383601, 0.03764448549066271,
0.039267163519703205]
valid loss: [0.10016259870358876, 0.09974808245897293, 0.09478138147720269,
0.10244000463613442, 0.09854166874928134, 0.08943230525723525,
0.0831080749630928, 0.08542938424008233, 0.0873507232006107,
0.08529418520629406, 0.08241834784192699, 0.07690527024013656,
0.0822777436780078, 0.07792759553662368, 0.07676779798098973,
0.07829846202262811, 0.08315108929361616, 0.0775779155748231,
0.08192909056586879, 0.08105260798973697, 0.08208648088787283,
0.08101476542651653, 0.0817071343106883, 0.07975080516189337,
0.08055217577410596]
```




[58]:
```
print(f'train margins: {train_errors}')
print(f'valid errors: {valid_errors}')
plt.figure(figsize=(15,6))
plt.plot(train_errors, label='train margin')
plt.plot(valid_errors, label='valid margin')
plt.xticks(range(len(train_losses)))
plt.title('Error Margin (years) for Model #2')
plt.legend()
plt.show()
```

```
train margins: [21.823564529418945, 22.339113235473633, 21.03468894958496,
22.901315689086914, 21.91028594970703, 19.1782283782959, 18.599281311035156,
19.1848087310791, 19.002391815185547, 18.273324966430664, 18.151315689086914,
16.695573806762695, 17.44078826904297, 16.084928512573242, 16.38815689086914,
16.72787094116211, 17.227272033691406, 15.983851432800293, 16.452152252197266,
16.26614761352539, 16.12858772277832, 15.765549659729004, 15.647727012634277,
15.066985130310059, 14.869616508483887]
valid errors: [24.942583084106445, 24.755979537963867, 24.248802185058594,
25.32535743713379, 24.808610916137695, 23.84688949584961, 23.167463302612305,
23.28229522705078, 23.665071487426758, 23.69377899169922, 23.311004638671875,
23.248802185058594, 23.64114761352539, 23.799041748046875, 23.401912689208984,
23.583730697631836, 24.01913833618164, 24.01913833618164, 24.46889877319336,
24.444974899291992, 24.746410369873047, 24.86602783203125, 25.1722469329834,
25.4401912689209, 26.258371353149414]
```




[59]:

```
for item in test_dataloader:
    text, normal_year, mask, real_year = item
    pred = continuous_model(text, mask)
    print(normal_year)
    print(pred)
    real_pred = (pred * GLOBAL_YEAR_RANGE) + GLOBAL_MIN_YEAR
    print(real_year)
    print(real_pred)
    break
```

```
tensor([[0.0909],
        [0.9394],
        [0.8384],
        [0.8384],
        [0.0808],
        [0.7980],
        [0.8081],
        [0.8384],
        [0.8384],
        [0.8586],
        [0.0808],
        [0.9091],
        [0.8384],
        [0.7980],
        [0.8889],
        [0.8384]], device='cuda:0')
tensor([[0.3355],
        [0.5345],
        [0.6820],
        [0.6521],
        [0.4220],
```

```
        [0.4893],
        [0.7072],
        [0.6164],
        [0.5855],
        [0.8129],
        [0.2428],
        [0.5030],
        [0.5720],
        [0.6129],
        [0.5706],
        [0.6360]], device='cuda:0', grad_fn=<AddmmBackward0>)
tensor([[1859.],
        [1943.],
        [1933.],
        [1933.],
        [1858.],
        [1929.],
        [1930.],
        [1933.],
        [1933.],
        [1935.],
        [1858.],
        [1940.],
        [1933.],
        [1929.],
        [1938.],
        [1933.]], device='cuda:0')
tensor([[1883.2115],
        [1902.9113],
        [1917.5160],
        [1914.5565],
        [1891.7797],
        [1898.4437],
        [1920.0087],
        [1911.0197],
        [1907.9668],
        [1930.4769],
        [1874.0419],
        [1899.7959],
        [1906.6312],
        [1910.6752],
        [1906.4899],
        [1912.9601]], device='cuda:0', grad_fn=<AddBackward0>)
```

### 5.3 I-BERT Model

[60]:
```
# ibert_model = DiscreteModel(ibert_submodel).to(device)
# ibert_model = DiscreteModel(ibert_submodel, flatten_all=True, n_layers=3).
 ↪to(device)
ibert_model = DiscreteModel(ibert_submodel, dropout=0.2, flatten_all=False).
 ↪to(device)
ibert_model.apply(init_classification_head_weights)
ibert_model.to(device)
print(f'Total trainable params: {count_params(ibert_model):,}')
print(f'          Total params: {total_params(ibert_model):,}')
print('IBert model initialized')
```

```
Total trainable params: 76,131
          Total params: 124,721,763
IBert model initialized
```

[61]:
```
BATCH_SIZE = 10
LR = 1e-3
N_EPOCHS = 10
CLIP = 1.0
```

[62]:
```
train_dataloader = DataLoader(
  train_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(discrete_collate, tokenizer=ibert_tokenizer),
  shuffle=False
)
val_dataloader = DataLoader(
  val_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(discrete_collate, tokenizer=ibert_tokenizer)
)
test_dataloader = DataLoader(
  test_dataset,
  batch_size=BATCH_SIZE,
  collate_fn=partial(discrete_collate, tokenizer=ibert_tokenizer)
)

optimizer = optim.Adam(ibert_model.parameters(), lr=LR)
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=10,␣
 ↪num_training_steps=N_EPOCHS*len(train_dataloader))
continuous_criterion = torch.nn.MSELoss()
```

[63]:
```
# Set model to fine-tune distil-bert
# for param in ibert_model.parameters():
#   param.requires_grad = True
```

```python
print(f'The model has {count_params(ibert_model):,} trainable parameters')
print(f'Total num of params: {total_params(ibert_model):,}')
print(ibert_model.enc_hid_dim)
# print(ibert_model)
```

```
The model has 76,131 trainable parameters
Total num of params: 124,721,763
768
```

[64]:
```python
train_loss = evaluate(ibert_model, train_dataloader, continuous_criterion,␣
 ↪device)
train_error = evaluate_dist(ibert_model, train_dataloader, device)

valid_loss = evaluate(ibert_model, val_dataloader, continuous_criterion, device)
valid_error = evaluate_dist(ibert_model, val_dataloader, device)

print(f'Initial Train Loss: {train_loss:.5f}')
print(f'Initial Train Error Margin (years): {train_error:.3f}')
print(f'Initial Valid Loss: {valid_loss:.5f}')
print(f'Initial Valid Error Margin (years): {valid_error:.3f}')

train_losses = []
train_errors = []
valid_losses = []
valid_errors = []
for epoch in range(N_EPOCHS):
  start_time = time.time()
  train_loss = train(ibert_model, train_dataloader, optimizer,␣
 ↪continuous_criterion, device, CLIP, scheduler)
  end_time = time.time()
  train_error = evaluate_dist(ibert_model, train_dataloader, device)

  valid_loss = evaluate(ibert_model, val_dataloader, continuous_criterion,␣
 ↪device)
  valid_error = evaluate_dist(ibert_model, val_dataloader, device)

  train_losses.append(train_loss)
  train_errors.append(train_error.item())
  valid_losses.append(valid_loss)
  valid_errors.append(valid_error.item())

  epoch_secs = end_time - start_time

  print(f'Epoch: {epoch+1:02} / {N_EPOCHS:02} | Time: {epoch_secs}s')
  print(f'\tTrain Loss: {train_loss:.5f}')
  print(f'\tTrain Error Margin: {train_error:.3f}')
```

```python
    print(f'\tValid Loss: {valid_loss:.5f}')
    print(f'\tValid Error Margin: {valid_error:.3f}')

test_loss = evaluate(ibert_model, test_dataloader, continuous_criterion, device)
test_error = evaluate_dist(ibert_model, test_dataloader, device)
print(f'TESTING:')
print(f'\tTesting Loss: {test_loss:.5f}')
print(f'\tTesting Error Margin: {test_error:.3f}')
```

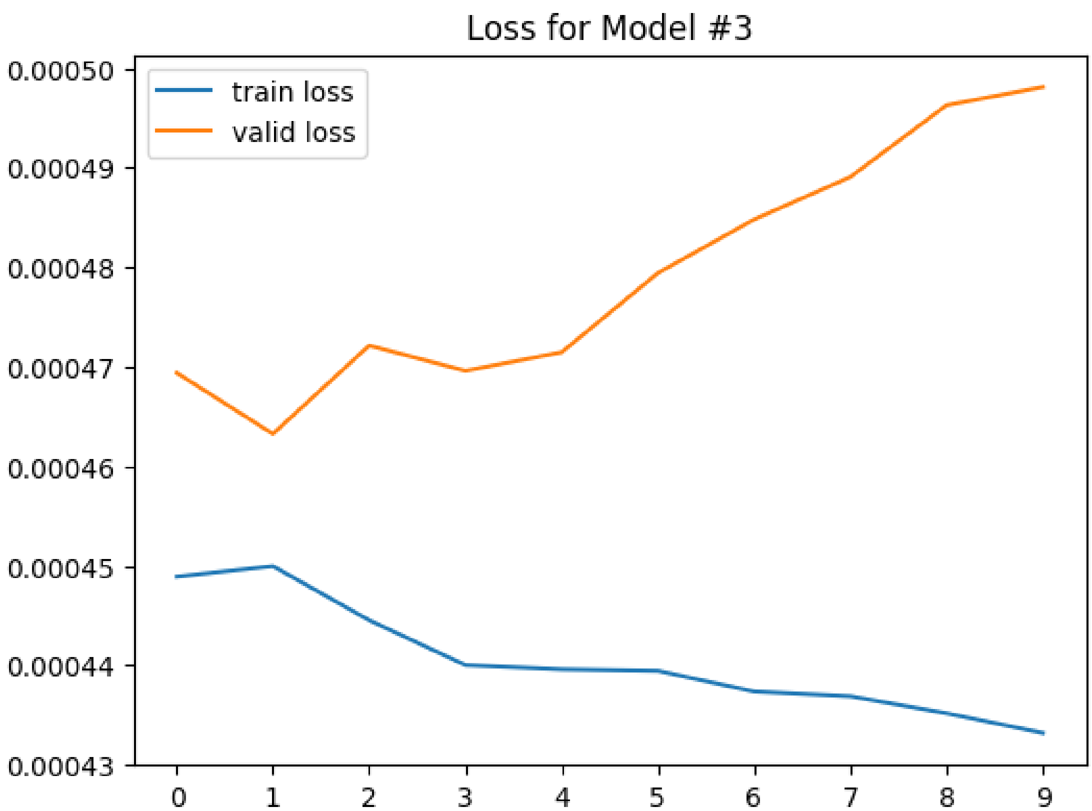
```
  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Initial Train Loss: 0.00047
Initial Train Error Margin (years): 37.177
Initial Valid Loss: 0.00047
Initial Valid Error Margin (years): 42.038

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 01 / 10 | Time: 56.411447048187256s
        Train Loss: 0.00045
        Train Error Margin: 32.944
        Valid Loss: 0.00047
        Valid Error Margin: 38.469

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 02 / 10 | Time: 56.42842936515808s
        Train Loss: 0.00045
        Train Error Margin: 29.523
        Valid Loss: 0.00046
        Valid Error Margin: 36.569

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]
```

```
  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 03 / 10 | Time: 56.46572709083557s
        Train Loss: 0.00044
        Train Error Margin: 21.418
        Valid Loss: 0.00047
        Valid Error Margin: 34.464

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 04 / 10 | Time: 56.37988471984863s
        Train Loss: 0.00044
        Train Error Margin: 22.402
        Valid Loss: 0.00047
        Valid Error Margin: 35.010

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 05 / 10 | Time: 56.381306409835815s
        Train Loss: 0.00044
        Train Error Margin: 20.553
        Valid Loss: 0.00047
        Valid Error Margin: 35.330

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 06 / 10 | Time: 56.528419733047485s
        Train Loss: 0.00044
        Train Error Margin: 19.025
        Valid Loss: 0.00048
        Valid Error Margin: 33.148

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]
```

```
Epoch: 07 / 10 | Time: 56.44230794906616s
        Train Loss: 0.00044
        Train Error Margin: 18.858
        Valid Loss: 0.00048
        Valid Error Margin: 33.737

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 08 / 10 | Time: 56.373910427093506s
        Train Loss: 0.00044
        Train Error Margin: 18.643
        Valid Loss: 0.00049
        Valid Error Margin: 33.440

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 09 / 10 | Time: 56.409791231155396s
        Train Loss: 0.00044
        Train Error Margin: 18.815
        Valid Loss: 0.00050
        Valid Error Margin: 33.383

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/168 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

Epoch: 10 / 10 | Time: 56.51579165458679s
        Train Loss: 0.00043
        Train Error Margin: 18.766
        Valid Loss: 0.00050
        Valid Error Margin: 32.818

  0%|          | 0/21 [00:00<?, ?it/s]

  0%|          | 0/21 [00:00<?, ?it/s]

TESTING:
        Testing Loss: 0.00054
        Testing Error Margin: 37.608
```

[68]:
```
print(f'train loss: {train_losses}')
print(f'valid loss: {valid_losses}')
plt.plot(train_losses, label='train loss')
plt.plot(valid_losses, label='valid loss')
plt.xticks(range(len(train_losses)))
plt.title('Loss for Model #3')
plt.legend()
plt.show()
```

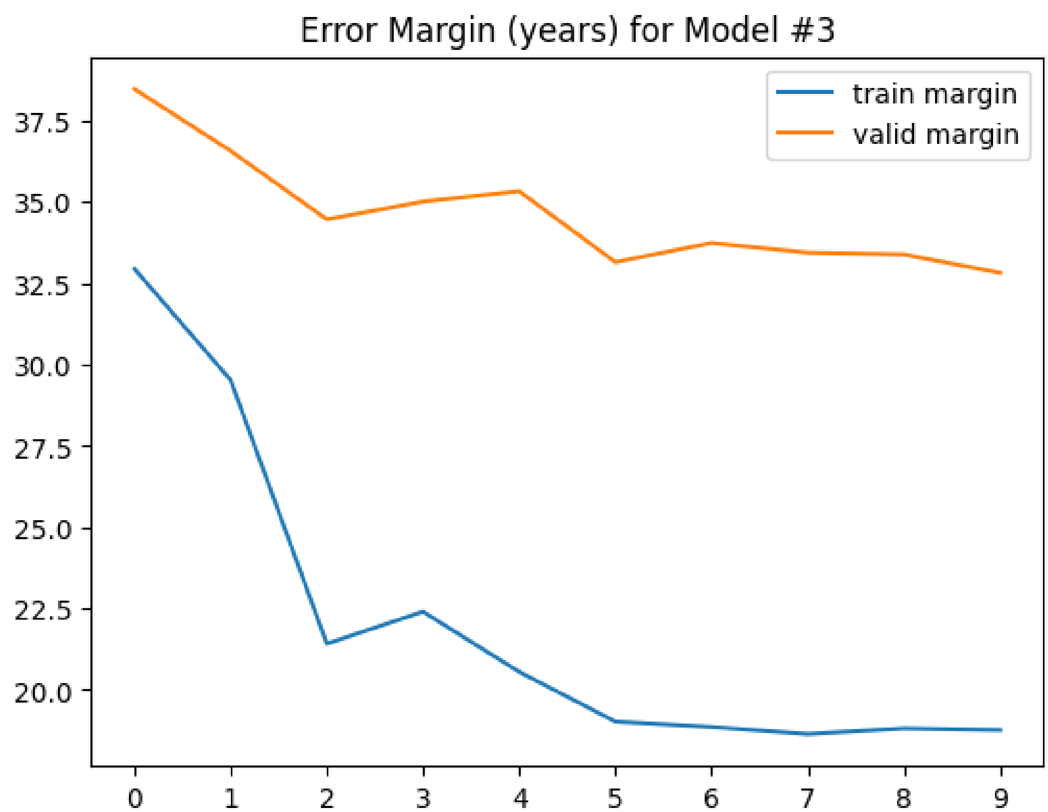
```
train loss: [0.0004488987937552987, 0.0004499630908650856,
0.0004445306305050118, 0.0004400197940412909, 0.00043959645803884735,
0.00043944393421822624, 0.00043736541175305667, 0.00043688581700691777,
0.00043517763766604255, 0.0004332087489837293]
valid loss: [0.0004693809051310555, 0.0004632440298640479 5,
0.0004721210931339079 6, 0.0004695839708715322 4, 0.0004714421063129391,
0.0004794025638451178 7, 0.0004847893932102514, 0.0004890521827508651,
0.0004962686064010043, 0.0004980776159624968]
```




[69]:
```
print(f'train margin: {train_errors}')
print(f'valid margin: {valid_errors}')
plt.plot(train_errors, label='train margin')
```

```
plt.plot(valid_errors, label='valid margin')
plt.xticks(range(len(train_losses)))
plt.title('Error Margin (years) for Model #3')
plt.legend()
plt.show()
```

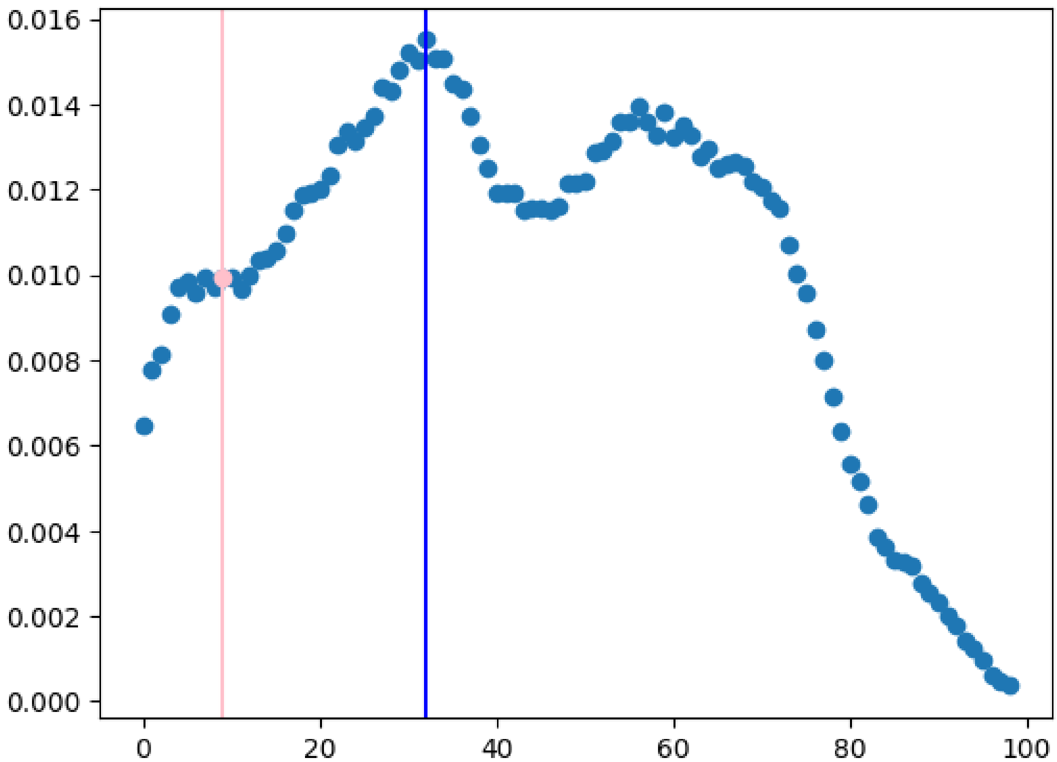
```
train margin: [32.94377899169922, 29.523324966430664, 21.418062210083008,
22.401912689208984, 20.552631378173828, 19.024520874023438, 18.858253479003906,
18.642942428588867, 18.8151912689209, 18.765548706054688]
valid margin: [38.46889877319336, 36.56937789916992, 34.464115142822266,
35.00956726074219, 35.330142974853516, 33.14832305908203, 33.736839294433594,
33.440189361572266, 33.382774353027344, 32.818180084228516]
```



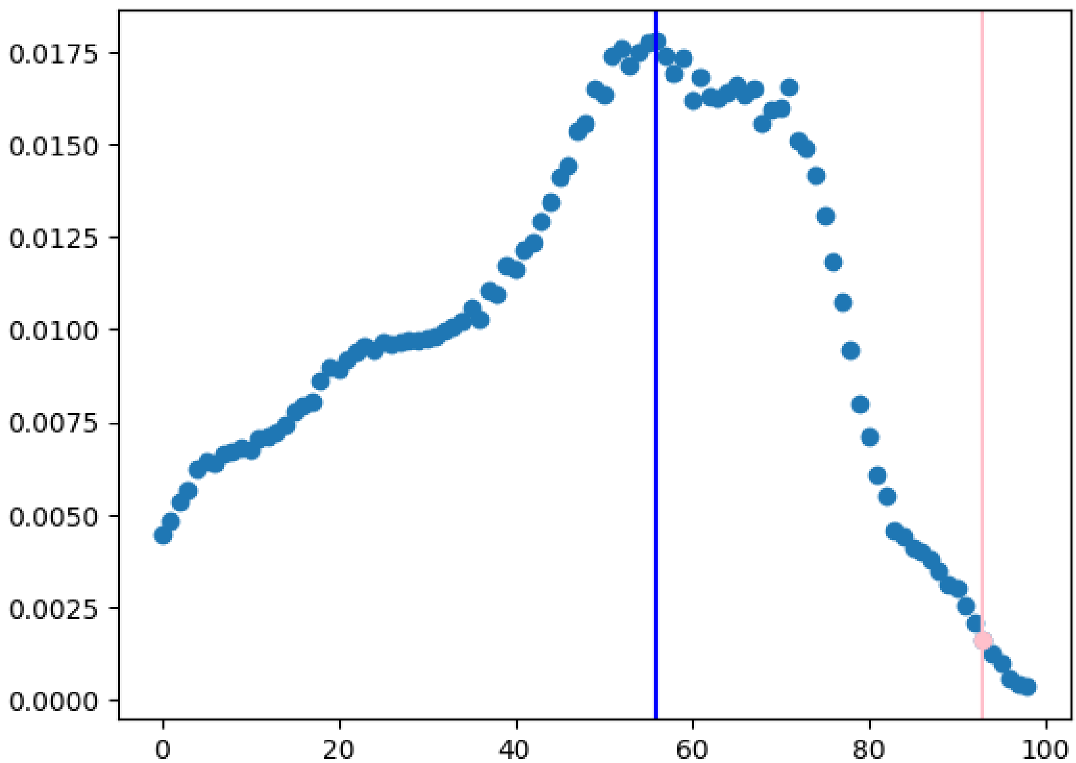

[67]:
```
# Just get one random batch from the dataloader
for item in test_dataloader:
  text, year, mask = item
  break

pred = ibert_model(text, mask)
for inputs, values in zip(year, pred):
  correct = torch.argmax(inputs).item()
```

```python
correct_guess_val = values.tolist()[correct]
guess = torch.argmax(values).item()
guess_val = values.tolist()[guess]
print(f'correct: {correct}')
print(f'argmax: {guess}')

X = torch.arange(GLOBAL_YEAR_RANGE).tolist()
plt.scatter(X, values.tolist())
plt.axvline(x=guess, color='blue')
plt.axvline(x=correct, color='pink')
plt.scatter(correct, correct_guess_val, color='pink')
plt.show()
```

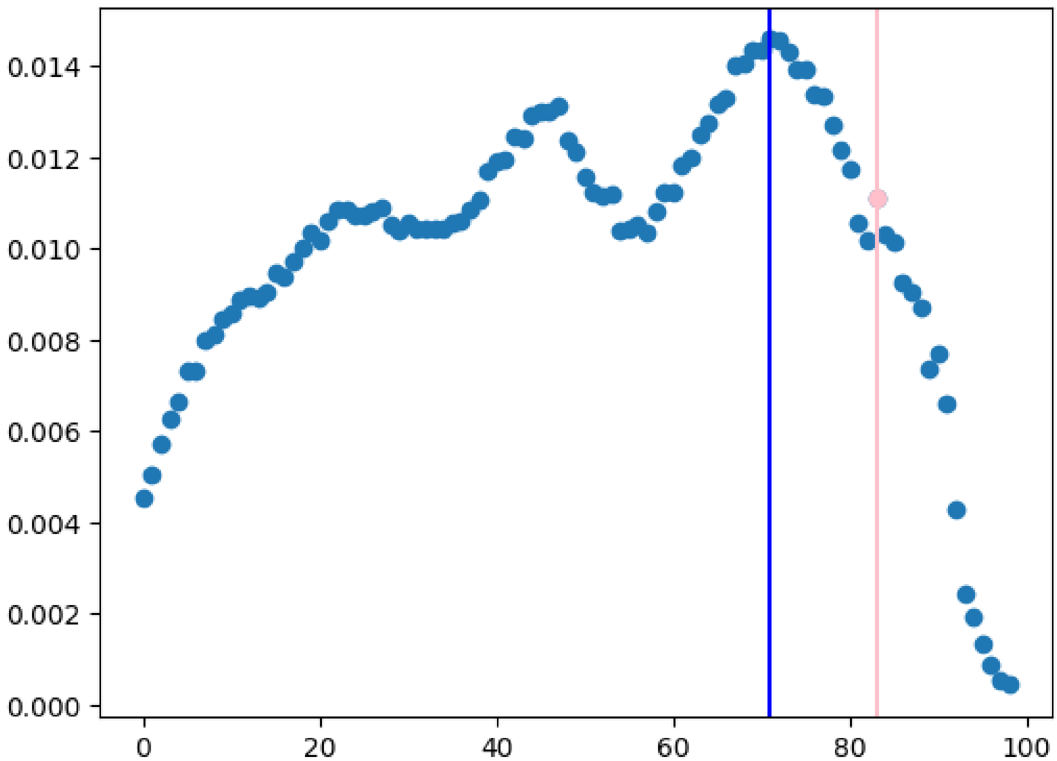
```
correct: 9
argmax: 32
```



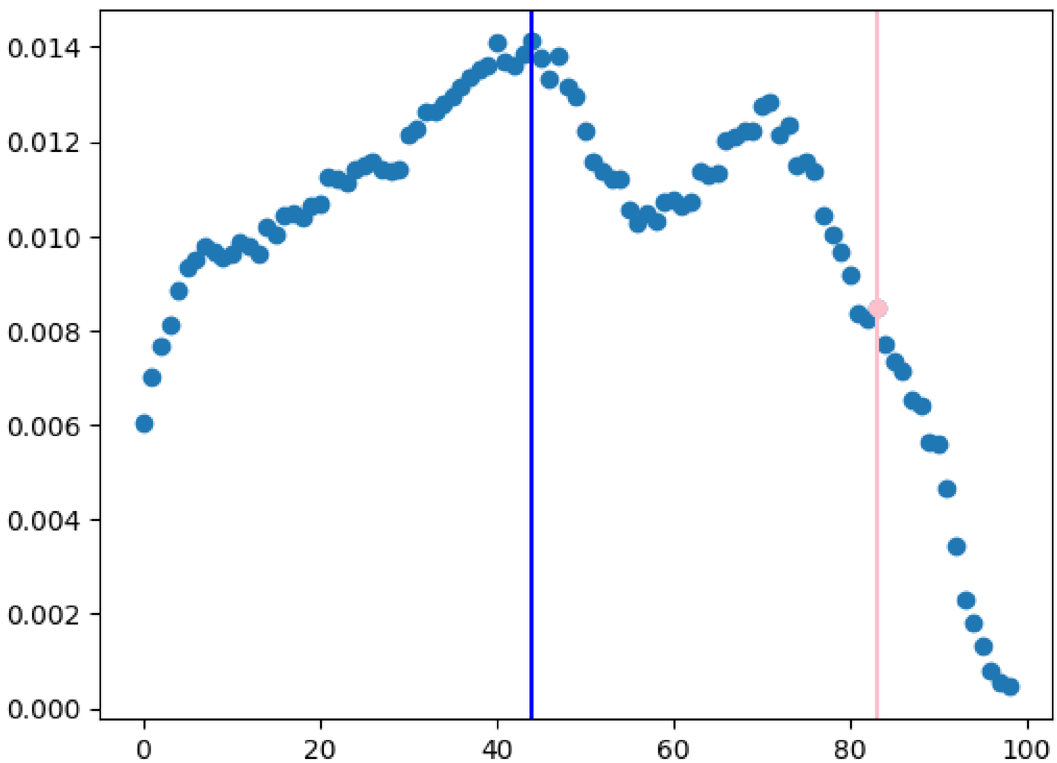

```
correct: 93
argmax: 56
```



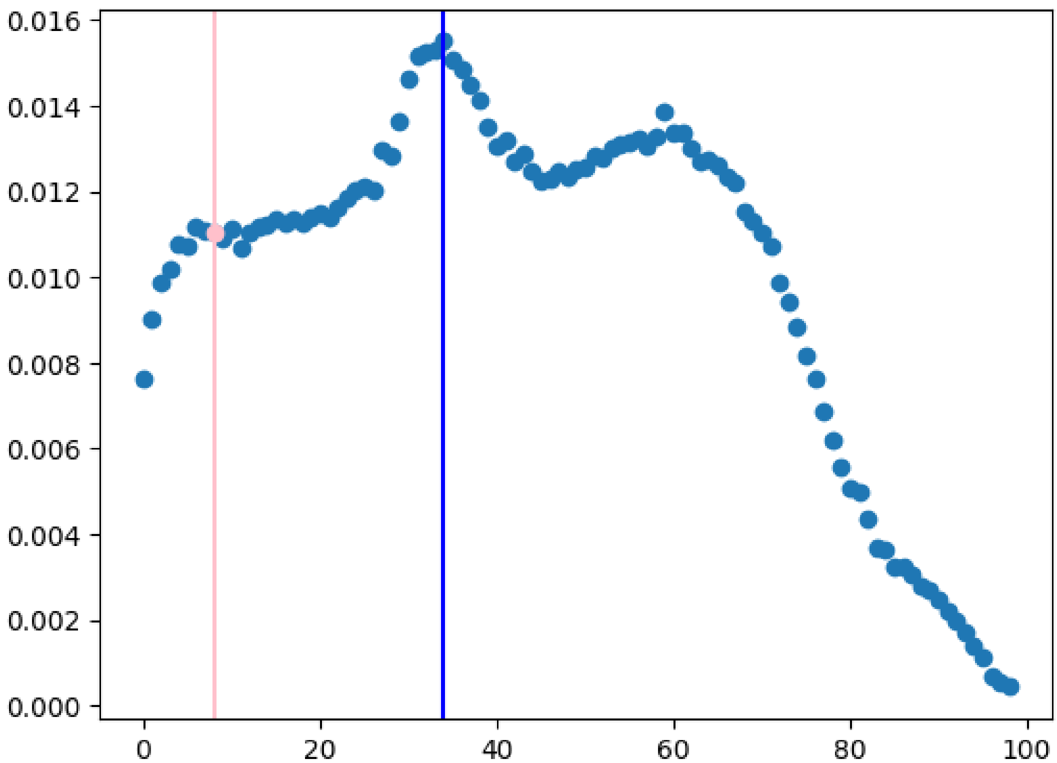

```
correct: 83
argmax: 71
```



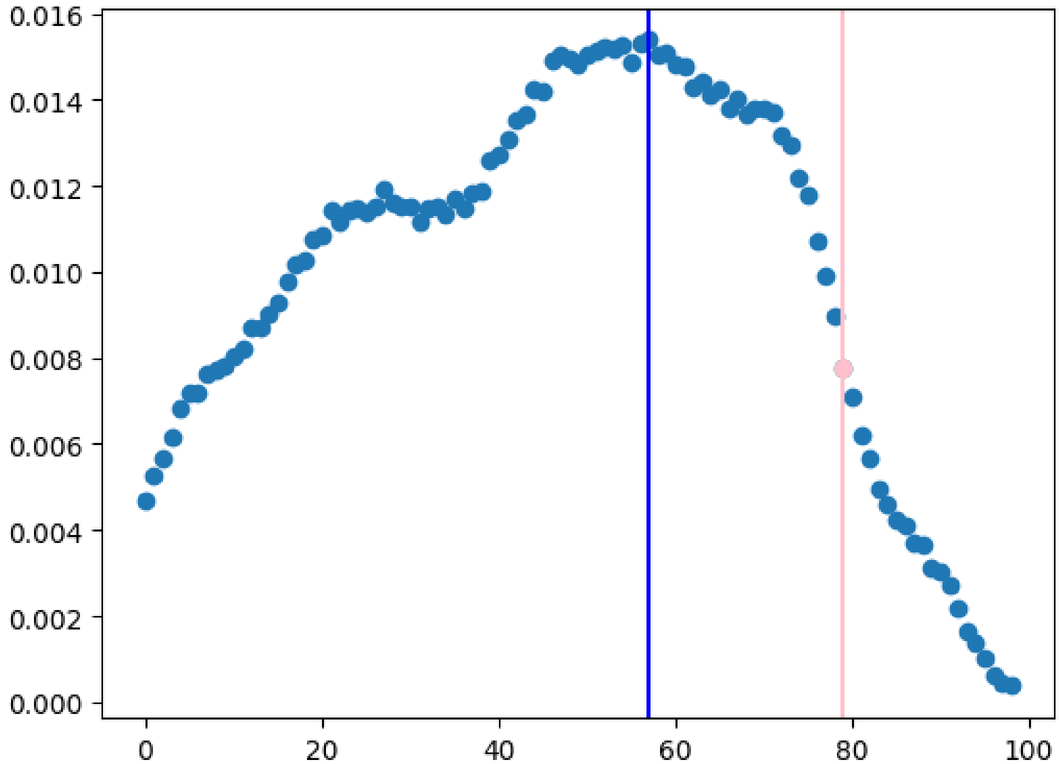

```
correct: 83
argmax: 44
```



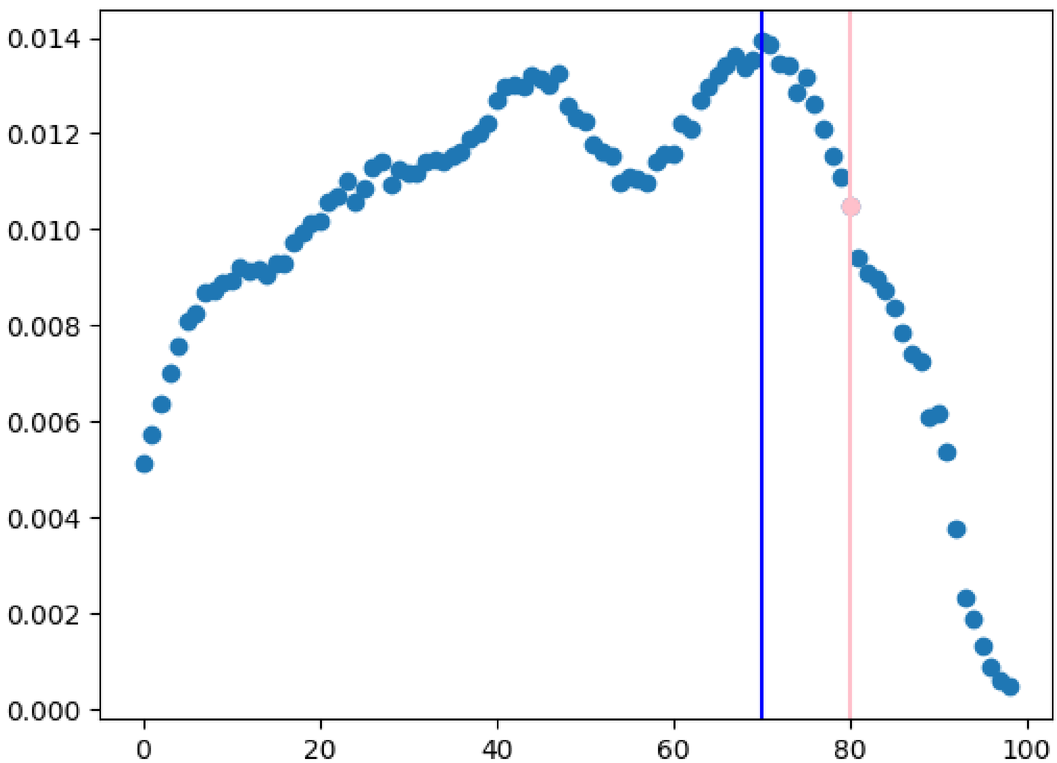

```
correct: 8
argmax: 34
```



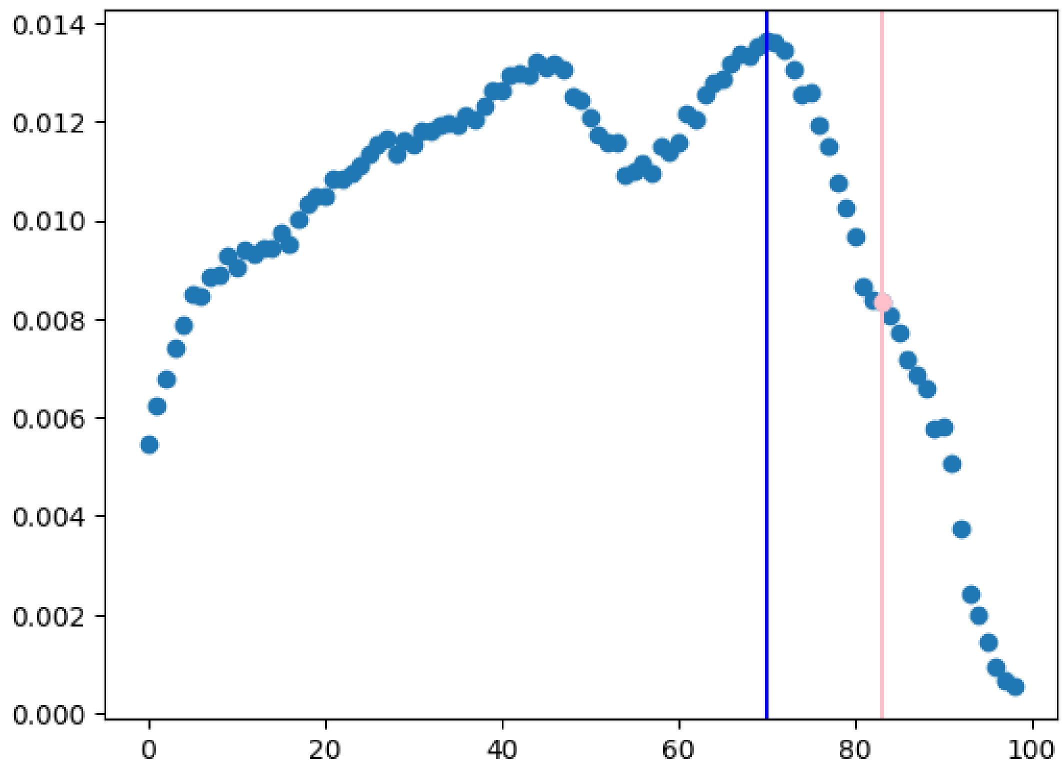

```
correct: 79
argmax: 57
```



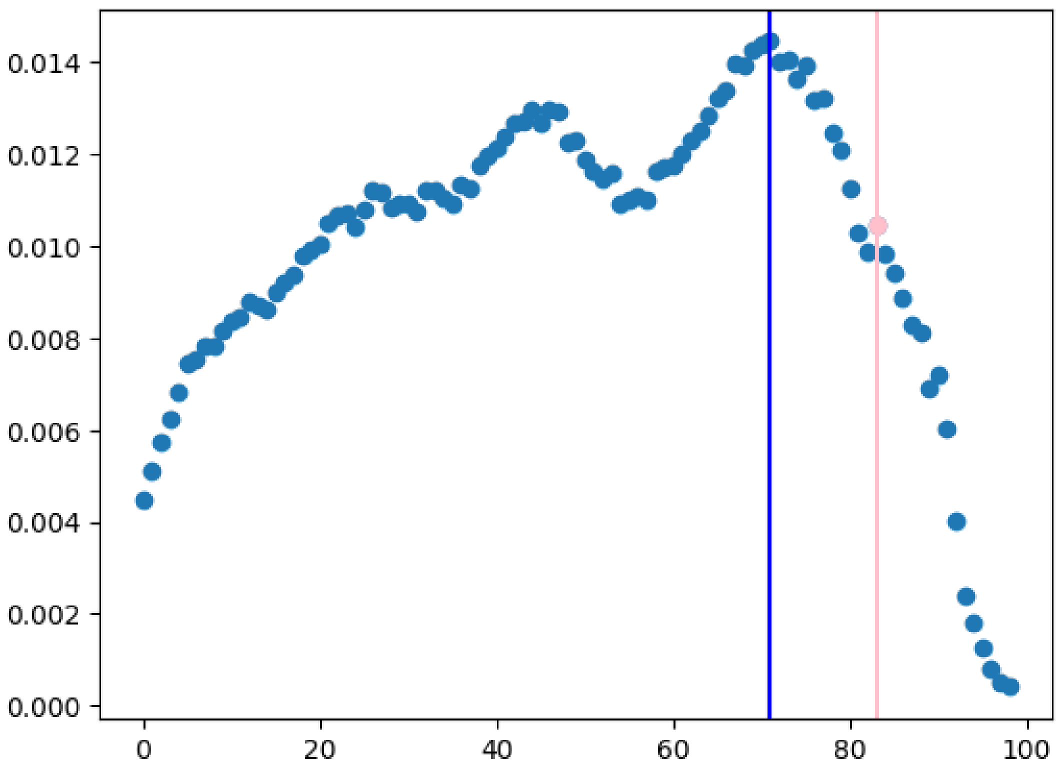

```
correct: 80
argmax: 70
```



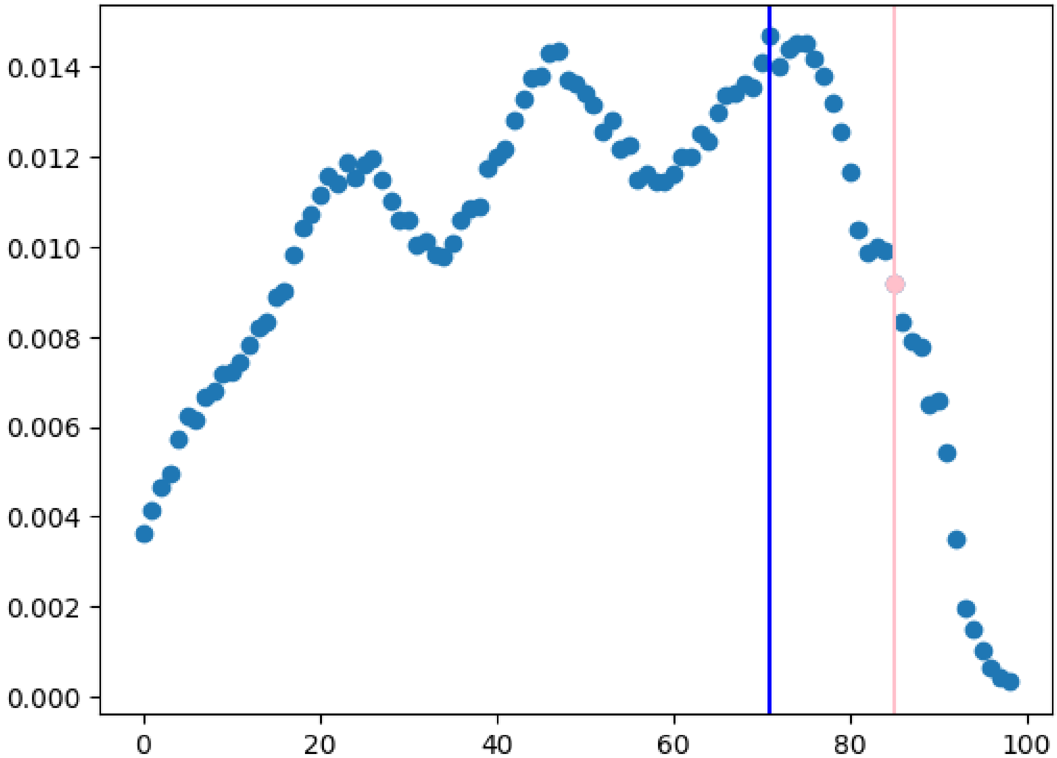

```
correct: 83
argmax: 70
```




```
correct: 83
argmax: 71
```




```
correct: 85
argmax: 71
```


0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
20
40
60
80
100