project_6_starter

April 22, 2023

1 Project 6: Analyzing Stock Sentiment from Twits

1.1 Instructions

Each problem consists of a function to implement and instructions on how to implement the function. The parts of the function that need to be implemented are marked with a `# TODO` comment.

1.2 Packages

When you implement the functions, you'll only need to you use the packages you've used in the classroom, like Pandas and Numpy. These packages will be imported for you. We recommend you don't add any import statements, otherwise the grader might not be able to run your code.

1.2.1 Load Packages

```
In [1]: import json
        import nltk
        import os
        import random
        import re
        import torch

        from torch import nn, optim
        import torch.nn.functional as F
```

1.3 Introduction

When deciding the value of a company, it's important to follow the news. For example, a product recall or natural disaster in a company's product chain. You want to be able to turn this information into a signal. Currently, the best tool for the job is a Neural Network.

For this project, you'll use posts from the social media site StockTwits. The community on StockTwits is full of investors, traders, and entrepreneurs. Each message posted is called a Twit. This is similar to Twitter's version of a post, called a Tweet. You'll build a model around these twits that generate a sentiment score.

We've collected a bunch of twits, then hand labeled the sentiment of each. To capture the degree of sentiment, we'll use a five-point scale: very negative, negative, neutral, positive, very positive. Each twit is labeled -2 to 2 in steps of 1, from very negative to very positive respectively.

You'll build a sentiment analysis model that will learn to assign sentiment to twits on its own, using this labeled data.

The first thing we should to do, is load the data.

1.4 Import Twits

1.4.1 Load Twits Data

This JSON file contains a list of objects for each twit in the `'data'` field:

```
{'data':
  {'message_body': 'Neutral twit body text here',
   'sentiment': 0},
  {'message_body': 'Happy twit body text here',
   'sentiment': 1},
   ...
}
```

The fields represent the following:

- `'message_body'`: The text of the twit.
- `'sentiment'`: Sentiment score for the twit, ranges from -2 to 2 in steps of 1, with 0 being neutral.

To see what the data look like by printing the first 10 twits from the list.

```
In [2]: with open(os.path.join('..', '..', 'data', 'project_6_stocktwits', 'twits.json'), 'r') a
            twits = json.load(f)

        print(twits['data'][:10])
```

```
[{'message_body': '$FITB great buy at 26.00...ill wait', 'sentiment': 2, 'timestamp': '2018-07-0
```

1.4.2 Length of Data

Now let's look at the number of twits in dataset. Print the number of twits below.

```
In [3]: """print out the number of twits"""

        # TODO Implement
        len(twits['data'])

Out[3]: 1548010
```

1.4.3 Split Message Body and Sentiment Score

```
In [4]: messages = [twit['message_body'] for twit in twits['data']]
        # Since the sentiment scores are discrete, we'll scale the sentiments to 0 to 4 for use
        sentiments = [twit['sentiment'] + 2 for twit in twits['data']]
```

1.5 Preprocessing the Data

With our data in hand we need to preprocess our text. These twits are collected by filtering on ticker symbols where these are denoted with a leader $ symbol in the twit itself. For example,

`{'message_body': 'RT @google Our annual look at the year in Google blogging (and beyond) http://t.co/sptHOAh8 $GOOG', 'sentiment': 0}`

The ticker symbols don't provide information on the sentiment, and they are in every twit, so we should remove them. This twit also has the `@google` username, again not providing sentiment information, so we should also remove it. We also see a URL `http://t.co/sptHOAh8`. Let's remove these too.

The easiest way to remove specific words or phrases is with regex using the `re` module. You can sub out specific patterns with a space:

```
re.sub(pattern, ' ', text)
```

This will substitute a space with anywhere the pattern matches in the text. Later when we tokenize the text, we'll split appropriately on those spaces.

1.5.1 Pre-Processing

```
In [5]: nltk.download('wordnet')


        def preprocess(message):
            """
            This function takes a string as input, then performs these operations:
                - lowercase
                - remove URLs
                - remove ticker symbols
                - removes punctuation
                - tokenize by splitting the string on whitespace
                - removes any single character tokens

            Parameters
            ----------
                message : The text message to be preprocessed.

            Returns
            -------
                tokens: The preprocessed text into tokens.
            """
            #TODO: Implement

            # Lowercase the twit message
            text = message.lower()

            # Replace URLs with a space in the message
            text = re.sub(r'http[s]?[\S]*', ' ', text)
        # Replace ticker symbols with a space. The ticker symbols are any stock symbol that
        text = re.sub(r'\$[\S]*', ' ', text)

        # Replace StockTwits usernames with a space. The usernames are any word that starts
        text = re.sub(r'@[\S]*', ' ', text)

        # Replace everything not a letter with a space
        text = re.sub(r'[^a-z]', ' ', text)

        # Tokenize by splitting the string on whitespace into a list of words
        tokens = text.split()

        # Lemmatize words using the WordNetLemmatizer. You can ignore any word that is not l
        wnl = nltk.stem.WordNetLemmatizer()
        tokens = [wnl.lemmatize(word) for word in tokens if len(word) > 1]


        assert type(tokens) == list, 'Tokens should be list'
        return tokens
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
```

Note: You must ensure that after preprocessing the text should NOT include: - Numbers - URLs - Single character tokens - Ticker symbols (these should be removed even if they don't appear at the beginning)

1.5.2 Preprocess All the Twits

Now we can preprocess each of the twits in our dataset. Apply the function `preprocess` to all the twit messages.

```
In [6]: # TODO Implement

        tokenized = [preprocess(message) for message in messages]
```

1.5.3 Bag of Words

Now with all of our messages tokenized, we want to create a vocabulary and count up how often each word appears in our entire corpus. Use the `Counter` function to count up all the tokens.

```
In [7]: from collections import Counter


        """
        Create a vocabulary by using Bag of words
        """


        # TODO: Implement
```

```
bow = Counter()
for words in tokenized:
    bow.update(words)
```

1.5.4 Frequency of Words Appearing in Message

With our vocabulary, now we'll remove some of the most common words such as 'the', 'and', 'it', etc. These words don't contribute to identifying sentiment and are really common, resulting in a lot of noise in our input. If we can filter these out, then our network should have an easier time learning.

We also want to remove really rare words that show up in a only a few twits. Here you'll want to divide the count of each word by the **number of messages** calculated in the code block above (i.e. `len(messages)`). Then remove words that only appear in some small fraction of the messages.

> Note: There is not an exact number for low and high-frequency cut-offs, however there is a correct optimal range. You should ideally set up low-frequency cut-off from 0.0000002 to 0.000007 (inclusive) and high-frequency from 5 to 20 (inclusive). If the number is too big, we lose lots of important words that we can use in our data.

```
In [8]: """
        Set the following variables:
            freqs
            low_cutoff
            high_cutoff
            K_most_common
        """

        # TODO Implement

        # Dictionart that contains the Frequency of words appearing in messages.
        # The key is the token and the value is the frequency of that word in the corpus.
        freqs = {w: f/len(tokenized) for w, f in bow.items()}

        # Float that is the frequency cutoff. Drop words with a frequency that is lower or equal
        low_cutoff = 0.000005

        # Integer that is the cut off for most common words. Drop words that are the `high_cutof
        high_cutoff = 15

        # The k most common words in the corpus. Use `high_cutoff` as the k.
        K_most_common = dict(bow.most_common(high_cutoff))

        filtered_words = [word for word in freqs if (freqs[word] > low_cutoff and word not in K_
        print(K_most_common)
        len(filtered_words)
```

```
{'the': 398753, 'to': 379487, 'is': 284865, 'for': 273537, 'on': 241663, 'of': 211334, 'and': 20
```

```
Out[8]: 21284

In [10]: {w: i for i, w in enumerate(filtered_words)}

Out[10]: {'great': 0,
          'buy': 1,
          'ill': 2,
          'wait': 3,
          'staanalystalert': 4,
          'jefferies': 5,
          'maintains': 6,
          'with': 7,
          'rating': 8,
          'hold': 9,
          'setting': 10,
          'target': 11,
          'price': 12,
          'usd': 13,
          'our': 14,
          'own': 15,
          'verdict': 16,
          'heard': 17,
          'there': 18,
          'guy': 19,
          'who': 20,
          'know': 21,
          'someone': 22,
          'think': 23,
          'somebody': 24,
          'something': 25,
          'stocktwits': 26,
          'reveal': 27,
          'yourself': 28,
          'why': 29,
          'drop': 30,
          'warren': 31,
          'buffet': 32,
          'taking': 33,
          'out': 34,
          'his': 35,
          'position': 36,
          'bear': 37,
          'have': 38,
          'reason': 39,
          'pay': 40,
          'more': 41,
          'attention': 42,
          'ok': 43,
'good': 44,
'we': 45,
're': 46,
'not': 47,
'dropping': 48,
'over': 49,
'weekend': 50,
'lol': 51,
'daily': 52,
'chart': 53,
'need': 54,
'get': 55,
'back': 56,
'above': 57,
'per': 58,
'week': 59,
'after': 60,
'spike': 61,
'if': 62,
'no': 63,
'news': 64,
'month': 65,
'bo': 66,
'then': 67,
'bingo': 68,
'what': 69,
'odds': 70,
'strong': 71,
'short': 72,
'ratio': 73,
'float': 74,
'via': 75,
'squeezing': 76,
'perfect': 77,
'place': 78,
'an': 79,
'option': 80,
'straddle': 81,
'near': 82,
'supporting': 83,
'trend': 84,
'start': 85,
'new': 86,
'monday': 87,
'expect': 88,
'volume': 89,
'across': 90,
'key': 91,
'company': 92,
'various': 93,
'sector': 94,
'breakout': 95,
'strategy': 96,
'current': 97,
'portfolio': 98,
'bull': 99,
'catalyst': 100,
'continuing': 101,
'uptrend': 102,
'pill': 103,
'pack': 104,
'amazon': 105,
'prime': 106,
'day': 107,
'earnings': 108,
'test': 109,
'break': 110,
'soon': 111,
'ha': 112,
'moved': 113,
'check': 114,
'movement': 115,
'peer': 116,
'mkm': 117,
'partner': 118,
'set': 119,
'too': 120,
'early': 121,
'tell': 122,
'going': 123,
'happen': 124,
'even': 125,
'about': 126,
'output': 127,
'still': 128,
'so': 129,
'many': 130,
'financially': 131,
'healthy': 132,
'problem': 133,
'meeting': 134,
'obligation': 135,
'bullish': 136,
'stock': 137,
'watch': 138,
'setup': 139,
'timeframes': 140,
'blog': 141,
'by': 142,
'just': 143,
'high': 144,
'alert': 145,
'next': 146,
'gt': 147,
'breakdown': 148,
'much': 149,
'better': 150,
'than': 151,
'industry': 152,
'average': 153,
'miss': 154,
'these': 155,
'confirms': 156,
'see': 157,
'violent': 158,
'move': 159,
'downward': 160,
'my': 161,
'base': 162,
'gap': 163,
'ma': 164,
'cabot': 165,
'weekly': 166,
'video': 167,
'quot': 168,
'resilient': 169,
'growth': 170,
'when': 171,
'wa': 172,
'kid': 173,
'told': 174,
'me': 175,
'moon': 176,
'made': 177,
'cheese': 178,
'percent': 179,
'rank': 180,
'th': 181,
'percentile': 182,
'thank': 183,
'don': 184,
'feel': 185,
'or': 186,
'do': 187,
'well': 188,
'they': 189,
'already': 190,
'during': 191,
'youtube': 192,
'interview': 193,
'mitch': 194,
'couldn': 195,
'remember': 196,
'single': 197,
'movie': 198,
'he': 199,
'watched': 200,
'all': 201,
'year': 202,
'except': 203,
'star': 204,
'war': 205,
'black': 206,
'panther': 207,
'clothes': 208,
'downgrade': 209,
'sale': 210,
'interest': 211,
'return': 212,
'equity': 213,
'although': 214,
'market': 215,
'doing': 216,
'advanced': 217,
'a': 218,
'info': 219,
'july': 220,
'be': 221,
'most': 222,
'normal': 223,
'friday': 224,
'contract': 225,
'call': 226,
'put': 227,
'third': 228,
'design': 229,
'win': 230,
'timing': 231,
'match': 232,
'probably': 233,
'arm': 234,
'alcohol': 235,
'tobacco': 236,
'president': 237,
'trump': 238,
'tweet': 239,
'spoke': 240,
'king': 241,
'saudi': 242,
'arabia': 243,
'explained': 244,
'him': 245,
'that': 246,
'because': 247,
'turmoil': 248,
'amp': 249,
'today': 250,
'insight': 251,
'intc': 252,
'mama': 253,
'said': 254,
'ice': 255,
'cream': 256,
'finish': 257,
'due': 258,
'noticed': 259,
'last': 260,
'jedi': 261,
'stream': 262,
'love': 263,
'here': 264,
'go': 265,
'leave': 266,
'view': 267,
'look': 268,
'ema': 269,
'held': 270,
'stochastics': 271,
'turning': 272,
'spy': 273,
'qqq': 274,
'opposite': 275,
'forward': 276,
'pe': 277,
'valuation': 278,
'can': 279,
'described': 280,
'cheap': 281,
'saber': 282,
'capital': 283,
'presentation': 284,
'common': 285,
'denominator': 286,
'tencent': 287,
'facebook': 288,
'google': 289,
'how': 290,
'one': 291,
'performing': 292,
'crude': 293,
'petroleum': 294,
'natural': 295,
'gas': 296,
'pu': 297,
'simply': 298,
'havent': 299,
'lost': 300,
'any': 301,
'brown': 302,
'run': 303,
'tech': 304,
'mf': 305,
'educated': 306,
'their': 307,
'shit': 308,
'every': 309,
'former': 310,
'global': 311,
'manufacturing': 312,
'pfizer': 313,
'independent': 314,
'director': 315,
'nat': 316,
'may': 317,
'safest': 318,
'special': 319,
'limited': 320,
'offer': 321,
'june': 322,
'offering': 323,
'discount': 324,
'intro': 325,
'program': 326,
'below': 327,
'financials': 328,
'summary': 329,
'jeff': 330,
'gundlach': 331,
'sohn': 332,
'long': 333,
'thesis': 334,
'energy': 335,
'low': 336,
'peg': 337,
'which': 338,
'compensates': 339,
'indicates': 340,
'rather': 341,
'pres': 342,
'coming': 343,
'seems': 344,
'support': 345,
'under': 346,
'likely': 347,
'bullshit': 348,
'worthless': 349,
'mean': 350,
'fucking': 351,
'nothing': 352,
'macau': 353,
'usa': 354,
'take': 355,
'let': 356,
'where': 357,
'right': 358,
'now': 359,
'ugly': 360,
'without': 361,
'clear': 362,
'level': 363,
'term': 364,
'bearish': 365,
'll': 366,
'previously': 367,
'mo': 368,
'followed': 369,
'end': 370,
'another': 371,
'runup': 372,
'dropped': 373,
'smart': 374,
'money': 375,
'betting': 376,
'deal': 377,
'update': 378,
'technical': 379,
'bad': 380,
'doe': 381,
'present': 382,
'nice': 383,
'opportunity': 384,
'completely': 385,
'scale': 386,
'time': 387,
'gobble': 388,
'play': 389,
'aka': 390,
'favorite': 391,
'them': 392,
'bitty': 393,
'kiddy': 394,
'hand': 395,
'awesome': 396,
'concerned': 397,
'eow': 398,
'might': 399,
'wk': 400,
'throwback': 401,
'u': 402,
'advisor': 403,
'upgrade': 404,
'shorties': 405,
'dreaming': 406,
'depression': 407,
'somethin': 408,
'loll': 409,
'updated': 410,
'hormel': 411,
'food': 412,
'dividend': 413,
'analysis': 414,
'consumer': 415,
'defensive': 416,
'intel': 417,
'profitability': 418,
'safety': 419,
'score': 420,
'technology': 421,
'both': 422,
'positive': 423,
'looking': 424,
'mad': 425,
'report': 426,
'block': 427,
'person': 428,
'spam': 429,
'should': 430,
'open': 431,
'projection': 432,
'thats': 433,
'true': 434,
'prefer': 435,
'from': 436,
'turd': 437,
'wal': 438,
'mart': 439,
'aristocrat': 440,
'valueinvesting': 441,
'dvb': 442,
'aaamp': 443,
'marriage': 444,
'bank': 445,
'account': 446,
'dm': 447,
'won': 448,
'stop': 449,
'posting': 450,
'holding': 451,
'big': 452,
'calm': 453,
'down': 454,
'luck': 455,
'chase': 456,
'continues': 457,
'hopefully': 458,
'catch': 459,
'before': 460,
'lisa': 461,
'announces': 462,
'rd': 463,
'turned': 464,
'off': 465,
'trading': 466,
'maybe': 467,
'altogether': 468,
'manipulation': 469,
'very': 470,
'rich': 471,
'make': 472,
'poor': 473,
'fuk': 474,
'almost': 475,
'pct': 476,
'buyback': 477,
'got': 478,
'share': 479,
'though': 480,
'drip': 481,
'dumped': 482,
'obv': 483,
'but': 484,
'jeez': 485,
'bit': 486,
'done': 487,
'bounce': 488,
'heikin': 489,
'ashi': 490,
'candle': 491,
'confirmation': 492,
'liking': 493,
'wife': 494,
'tree': 495,
'fiddy': 496,
'sure': 497,
'piggy': 498,
'lot': 499,
'coin': 500,
'spend': 501,
'candy': 502,
'microsoft': 503,
'investor': 504,
'aren': 505,
'happy': 506,
'royal': 507,
'canada': 508,
'reiterates': 509,
'outperform': 510,
'loop': 511,
'raise': 512,
'never': 513,
'been': 514,
'wrong': 515,
'miracle': 516,
'period': 517,
'bollinger': 518,
'band': 519,
'dare': 520,
'say': 521,
'didn': 522,
'guilty': 523,
'fudge': 524,
'hardly': 525,
'ever': 526,
'informative': 527,
'comment': 528,
'page': 529,
'broken': 530,
'giving': 531,
'endless': 532,
'icon': 533,
'give': 534,
'viral': 535,
'walk': 536,
'away': 537,
'highlight': 538,
'growing': 539,
'democrat': 540,
'leaving': 541,
'party': 542,
'robert': 543,
'baird': 544,
'neutral': 545,
'weed': 546,
'other': 547,
'drug': 548,
'guess': 549,
'reasonable': 550,
'nike': 551,
'winning': 552,
'click': 553,
'booking': 554,
'app': 555,
'native': 556,
'seamless': 557,
'ap': 558,
'tapped': 559,
'acting': 560,
'reach': 561,
'iconic': 562,
'brand': 563,
'built': 564,
'mobile': 565,
'show': 566,
'morning': 567,
'heading': 568,
'ath': 569,
'acquisition': 570,
'total': 571,
'past': 572,
'exit': 573,
'zone': 574,
'fantastic': 575,
'read': 576,
'oil': 577,
'tighten': 578,
'getting': 579,
'closer': 580,
'greatest': 581,
'yearly': 582,
'performance': 583,
'did': 584,
'champion': 585,
'cashflow': 586,
'highyield': 587,
'pretty': 588,
'risky': 589,
'gamble': 590,
'either': 591,
'plummet': 592,
'rally': 593,
'stronger': 594,
'apple': 595,
'talking': 596,
'referring': 597,
'punish': 598,
'seller': 599,
'warehouse': 600,
'sense': 601,
'old': 602,
'gather': 603,
'dust': 604,
'pc': 605,
'gamers': 606,
'crypto': 607,
'miner': 608,
'gpu': 609,
'enthusiast': 610,
'waiting': 611,
'two': 612,
'only': 613,
'medium': 614,
'decent': 615,
'pattern': 616,
'five': 617,
'baba': 618,
'raytheon': 619,
'industrials': 620,
'military': 621,
'trending': 622,
'sooooon': 623,
'drunk': 624,
'doesn': 625,
'through': 626,
'gapping': 627,
'parker': 628,
'prep': 629,
'glass': 630,
'seen': 631,
'focus': 632,
'fashion': 633,
'wrt': 634,
'wearable': 635,
'like': 636,
'sbux': 637,
'paying': 638,
'full': 639,
'sex': 640,
'change': 641,
'employee': 642,
'hit': 643,
'bottom': 644,
'line': 645,
'recap': 646,
'since': 647,
'post': 648,
'trade': 649,
'included': 650,
'hsy': 651,
'uri': 652,
'sonc': 653,
'dri': 654,
'kmx': 655,
'wynn': 656,
'akam': 657,
'enterprise': 658,
'value': 659,
'v': 660,
'cap': 661,
'difference': 662,
'actual': 663,
'asset': 664,
'spread': 665,
'way': 666,
'undervalued': 667,
'million': 668,
'added': 669,
'dont': 670,
'become': 671,
'such': 672,
'monopoly': 673,
'dipping': 674,
'into': 675,
'everything': 676,
'airline': 677,
'possibility': 678,
'loss': 679,
'suggested': 680,
'chartmill': 681,
'analyzer': 682,
'resistance': 683,
'len': 684,
'reversal': 685,
'progress': 686,
've': 687,
'recently': 688,
'closed': 689,
'list': 690,
'bought': 691,
'same': 692,
'used': 693,
'according': 694,
'data': 695,
'reported': 696,
'finra': 697,
'clocked': 698,
'official': 699,
'came': 700,
'saying': 701,
'agree': 702,
'resume': 703,
'cover': 704,
'penis': 705,
'front': 706,
'leg': 707,
'would': 708,
'some': 709,
'am': 710,
'watching': 711,
'wonder': 712,
'valued': 713,
'inning': 714,
'higher': 715,
'also': 716,
'shorted': 717,
'consolidation': 718,
'cautious': 719,
'stunned': 720,
'deeply': 721,
'disappointed': 722,
'close': 723,
'ending': 724,
'killed': 725,
'best': 726,
'february': 727,
'funny': 728,
'thing': 729,
'similar': 730,
'story': 731,
'different': 732,
'crab': 733,
'measured': 734,
'divergence': 735,
'really': 736,
'momentum': 737,
'starting': 738,
'shi': 739,
'venture': 740,
'tariff': 741,
'increase': 742,
'profit': 743,
'recession': 744,
'come': 745,
'keep': 746,
'buying': 747,
'pure': 748,
'min': 749,
'indicating': 750,
'accumulation': 751,
'eod': 752,
'could': 753,
'pop': 754,
'heavy': 755,
'acn': 756,
'camp': 757,
'gi': 758,
'payx': 759,
'fdx': 760,
'goo': 761,
'squeeze': 762,
'pullback': 763,
'upside': 764,
'nvda': 765,
'safe': 766,
'yield': 767,
'div': 768,
'adding': 769,
'gladly': 770,
'irrationality': 771,
'reign': 772,
'possible': 773,
'excess': 774,
'gain': 775,
'welcome': 776,
'gl': 777,
'seeing': 778,
'jumping': 779,
'joy': 780,
'least': 781,
'given': 782,
'action': 783,
'bouncing': 784,
'continuation': 785,
'hopeful': 786,
'science': 787,
'officially': 788,
'became': 789,
'major': 790,
'university': 791,
'ai': 792,
'era': 793,
'second': 794,
'minute': 795,
'rule': 796,
'applies': 797,
'six': 798,
'fc': 799,
'fds': 800,
'avav': 801,
'killing': 802,
'tax': 803,
'cut': 804,
'idea': 805,
'haven': 806,
'posted': 807,
'while': 808,
'omg': 809,
'toy': 810,
'land': 811,
'amazing': 812,
'contributed': 813,
'ytd': 814,
'popular': 815,
'word': 816,
'buyin': 817,
'those': 818,
'dip': 819,
'haha': 820,
'makin': 821,
'rain': 822,
'white': 823,
'house': 824,
'quality': 825,
'moment': 826,
'doubt': 827,
'production': 828,
'eps': 829,
'constellation': 830,
'disney': 831,
'clinch': 832,
'acquire': 833,
'fox': 834,
'group': 835,
'counter': 836,
'aggregate': 837,
'southwest': 838,
'flight': 839,
'dallas': 840,
'delta': 841,
'spat': 842,
'california': 843,
'netflix': 844,
'inc': 845,
'nflx': 846,
'versus': 847,
'fb': 848,
'starbucks': 849,
'using': 850,
'iex': 851,
'calculate': 852,
'shorters': 853,
'bankruptcy': 854,
'emergency': 855,
'investment': 856,
'counsel': 857,
'lower': 858,
'stake': 859,
'unitedhealth': 860,
'unh': 861,
'vega': 862,
'laptop': 863,
'investing': 864,
'annuity': 865,
'yr': 866,
'bond': 867,
'continue': 868,
'add': 869,
'unsure': 870,
'country': 871,
'amd': 872,
'ryzen': 873,
'gettin': 874,
'blasted': 875,
'cx': 876,
'aaoi': 877,
'aa': 878,
'yy': 879,
'snx': 880,
'shark': 881,
'definitely': 882,
'august': 883,
'g': 884,
'point': 885,
'huge': 886,
'closing': 887,
'scalp': 888,
'giant': 889,
'marked': 890,
'compared': 891,
'book': 892,
'cheaply': 893,
'quite': 894,
'job': 895,
'started': 896,
'clearly': 897,
'cup': 898,
'holder': 899,
'inverse': 900,
'confidence': 901,
'debt': 902,
'sell': 903,
'interesting': 904,
'frame': 905,
'trader': 906,
'meanwhile': 907,
'galaxy': 908,
'far': 909,
'broadening': 910,
'wedge': 911,
'descending': 912,
'large': 913,
'im': 914,
'pt': 915,
'three': 916,
'falling': 917,
'peak': 918,
'fourth': 919,
'everyone': 920,
'coke': 921,
'smile': 922,
'thanks': 923,
'selling': 924,
'prove': 925,
'terrible': 926,
'error': 927,
'article': 928,
'hint': 929,
'sogou': 930,
'finger': 931,
'crossed': 932,
'triple': 933,
'top': 934,
'bro': 935,
'work': 936,
'delete': 937,
'blue': 938,
'arrow': 939,
'your': 940,
'delusion': 941,
'overbought': 942,
'changing': 943,
'mr': 944,
'yo': 945,
'hi': 946,
'bed': 947,
'night': 948,
'prepared': 949,
'policy': 950,
'store': 951,
'product': 952,
'fyi': 953,
'ramp': 954,
'worker': 955,
'revolt': 956,
'hill': 957,
'rail': 958,
'people': 959,
'manager': 960,
'circuit': 961,
'city': 962,
'fail': 963,
'despite': 964,
'bb': 965,
'incr': 966,
'thinking': 967,
'mm': 968,
'driving': 969,
'prior': 970,
'argument': 971,
'sound': 972,
'becoming': 973,
'loses': 974,
'gaming': 975,
'channel': 976,
'cash': 977,
'machine': 978,
'semi': 979,
'altria': 980,
'aflac': 981,
'graph': 982,
'tripple': 983,
'reached': 984,
'want': 985,
'vagina': 986,
'oled': 987,
'panel': 988,
'steady': 989,
'amid': 990,
'weak': 991,
'demand': 992,
'display': 993,
'leak': 994,
'foldable': 995,
'dual': 996,
'screen': 997,
'surface': 998,
'device': 999,
...}
```

1.5.5 Updating Vocabulary by Removing Filtered Words

Let's creat three variables that will help with our vocabulary.

```
In [11]: """
         Set the following variables:
             vocab
             id2vocab
             filtered
         """

         #TODO Implement

         # A dictionary for the `filtered_words`. The key is the word and value is an id that re
         vocab = {w: i for i, w in enumerate(filtered_words)}
         # Reverse of the `vocab` dictionary. The key is word id and value is the word.
         id2vocab = {w: i for w, i in enumerate(filtered_words)}
         # tokenized with the words not in `filtered_words` removed.
         filtered = [[word for word in message if word in vocab] for message in tokenized]

         assert set(vocab.keys()) == set(id2vocab.values()), 'Check vocab and id2vocab dictionar
```

1.5.6 Balancing the classes

Let's do a few last pre-processing steps. If we look at how our twits are labeled, we'll find that 50% of them are neutral. This means that our network will be 50% accurate just by guessing 0 every single time. To help our network learn appropriately, we'll want to balance our classes. That is, make sure each of our different sentiment scores show up roughly as frequently in the data.

What we can do here is go through each of our examples and randomly drop twits with neutral sentiment. What should be the probability we drop these twits if we want to get around 20% neutral twits starting at 50% neutral? We should also take this opportunity to remove messages with length 0.

```
In [12]: balanced = {'messages': [], 'sentiments':[]}

         n_neutral = sum(1 for each in sentiments if each == 2)
         N_examples = len(sentiments)
         keep_prob = (N_examples - n_neutral)/4/n_neutral

         for idx, sentiment in enumerate(sentiments):
             message = filtered[idx]
             if len(message) == 0:
                 # skip this message because it has length zero
                 continue
             elif sentiment != 2 or random.random() < keep_prob:
                 balanced['messages'].append(message)
                 balanced['sentiments'].append(sentiment)
```

If you did it correctly, you should see the following result

```
In [13]: n_neutral = sum(1 for each in balanced['sentiments'] if each == 2)
         N_examples = len(balanced['sentiments'])
         n_neutral/N_examples

Out[13]: 0.19428327147659483
```

Finally let's convert our tokens into integer ids which we can pass to the network.

```
In [14]: token_ids = [[vocab[word] for word in message] for message in balanced['messages']]
         sentiments = balanced['sentiments']
```

1.6 Neural Network

Now we have our vocabulary which means we can transform our tokens into ids, which are then passed to our network. So, let's define the network now!

Here is a nice diagram showing the network we'd like to build:

Embed -> RNN -> Dense -> Softmax

1.6.1 Implement the text classifier

Before we build text classifier, if you remember from the other network that you built in "Sentiment Analysis with an RNN" exercise - which there, the network called " SentimentRNN", here we named it "TextClassifer" - consists of three main parts: 1) init function `__init__` 2) forward pass `forward` 3) hidden state `init_hidden`.

This network is pretty similar to the network you built expect in the `forward` pass, we use softmax instead of sigmoid. The reason we are not using sigmoid is that the output of NN is not a binary. In our network, sentiment scores have 5 possible outcomes. We are looking for an outcome with the highest probability thus softmax is a better choice.

```
In [70]: class TextClassifier(nn.Module):
             def __init__(self, vocab_size, embed_size, lstm_size, output_size, lstm_layers=1, d
                 """
                 Initialize the model by setting up the layers.

                 Parameters
                 ----------
                     vocab_size : The vocabulary size.
                     embed_size : The embedding layer size.
                     lstm_size : The LSTM layer size.
                     output_size : The output size.
                     lstm_layers : The number of LSTM layers.
                     dropout : The dropout probability.
                 """

                 super().__init__()
                 self.vocab_size = vocab_size
                 self.embed_size = embed_size
                 self.lstm_size = lstm_size
```

```
        self.output_size = output_size
        self.lstm_layers = lstm_layers
        self.dropout = dropout

        # TODO Implement

        # Setup embedding layer
        self.embedding = nn.Embedding(num_embeddings=self.vocab_size, embedding_dim=sel

        # Setup additional layers
        self.lstm = nn.LSTM(
            embed_size,
            lstm_size,
            lstm_layers,
            dropout=dropout,
            batch_first=False
        )
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(lstm_size, output_size)
        self.log_softmax = nn.LogSoftmax(dim=1)


    def init_hidden(self, batch_size):
        """
        Initializes hidden state

        Parameters
        ----------
            batch_size : The size of batches.

        Returns
        -------
            hidden_state

        """

        # TODO Implement

        # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        return (weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_(),
                weight.new(self.lstm_layers, batch_size, self.lstm_size).zero_())

    def forward(self, nn_input, hidden_state):
        """
        Perform a forward pass of our model on nn_input.
```

```
            Parameters
            ----------
                nn_input : The batch of input to the NN.
                hidden_state : The LSTM hidden state.

            Returns
            -------
                logps: log softmax output
                hidden_state: The new hidden state.

            """

            # TODO Implement
            embed = self.embedding(nn_input)
            lstm_out, hidden_state = self.lstm(embed, hidden_state)

            lstm_out = lstm_out[-1]

            logps = self.log_softmax(self.dropout(self.linear(lstm_out)))

            return logps, hidden_state
```

1.6.2 View Model

```
In [71]: model = TextClassifier(len(vocab), 10, 6, 5, dropout=0.1, lstm_layers=2)
         model.embedding.weight.data.uniform_(-1, 1)
         input = torch.randint(0, 1000, (5, 4), dtype=torch.int64)
         hidden = model.init_hidden(4)

         logps, _ = model.forward(input, hidden)
         print(logps)
```

```
tensor([[-1.6877, -1.5960, -1.5960, -1.5865, -1.5847],
        [-1.6265, -1.3887, -1.9567, -1.6154, -1.5426],
        [-1.5764, -1.4089, -1.9749, -1.6067, -1.5628],
        [-1.6395, -1.3874, -1.9547, -1.6044, -1.5437]])
```

1.7 Training

1.7.1 DataLoaders and Batching

Now we should build a generator that we can use to loop through our data. It'll be more efficient if we can pass our sequences in as batches. Our input tensors should look like `(sequence_length, batch_size)`. So if our sequences are 40 tokens long and we pass in 25 sequences, then we'd have an input size of `(40, 25)`.

If we set our sequence length to 40, what do we do with messages that are more or less than 40 tokens? For messages with fewer than 40 tokens, we will pad the empty spots with zeros. We should be sure to **left** pad so that the RNN starts from nothing before going through the data. If the message has 20 tokens, then the first 20 spots of our 40 long sequence will be 0. If a message has more than 40 tokens, we'll just keep the first 40 tokens.

```
In [72]: def dataloader(messages, labels, sequence_length=30, batch_size=32, shuffle=False):
             """
             Build a dataloader.
             """
             if shuffle:
                 indices = list(range(len(messages)))
                 random.shuffle(indices)
                 messages = [messages[idx] for idx in indices]
                 labels = [labels[idx] for idx in indices]

             total_sequences = len(messages)

             for ii in range(0, total_sequences, batch_size):
                 batch_messages = messages[ii: ii+batch_size]

                 # First initialize a tensor of all zeros
                 batch = torch.zeros((sequence_length, len(batch_messages)), dtype=torch.int64)
                 for batch_num, tokens in enumerate(batch_messages):
                     token_tensor = torch.tensor(tokens)
                     # Left pad!
                     start_idx = max(sequence_length - len(token_tensor), 0)
                     batch[start_idx:, batch_num] = token_tensor[:sequence_length]

                 label_tensor = torch.tensor(labels[ii: ii+len(batch_messages)])

                 yield batch, label_tensor
```

1.7.2 Training and Validation

With our data in nice shape, we'll split it into training and validation sets.

```
In [73]: """
         Split data into training and validation datasets. Use an appropriate split size.
         The features are the `token_ids` and the labels are the `sentiments`.
         """

         # TODO Implement
         split_size = 0.8
         split_idx = int(len(token_ids)*split_size)

         train_features = token_ids[:split_idx]
         valid_features = token_ids[split_idx:]
         train_labels = sentiments[:split_idx]
         valid_labels = sentiments[split_idx:]
```

```
In [74]: text_batch, labels = next(iter(dataloader(train_features, train_labels, sequence_length
         model = TextClassifier(len(vocab)+1, 200, 128, 5, dropout=0.)
         hidden = model.init_hidden(64)
         logps, hidden = model.forward(text_batch, hidden)
```

1.7.3 Training

It's time to train the neural network!

```
In [75]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

         model = TextClassifier(len(vocab)+1, 1024, 512, 5, lstm_layers=2, dropout=0.2)
         model.embedding.weight.data.uniform_(-1, 1)
         model.to(device)
```

```
Out[75]: TextClassifier(
           (embedding): Embedding(21285, 1024)
           (lstm): LSTM(1024, 512, num_layers=2, dropout=0.2)
           (dropout): Dropout(p=0.2)
           (linear): Linear(in_features=512, out_features=5, bias=True)
           (log_softmax): LogSoftmax()
         )
```

```
In [76]: """
         Train your model with dropout. Make sure to clip your gradients.
         Print the training loss, validation loss, and validation accuracy for every 100 steps.
         """

         epochs = 2
         batch_size = 512
         sequence_length = 20
         learning_rate = 0.001
         clip = 5

         print_every = 100
         criterion = nn.NLLLoss()
         optimizer = optim.Adam(model.parameters(), lr=learning_rate)
         model.train()

         for epoch in range(epochs):
             print('Starting epoch {}'.format(epoch + 1))

             steps = 0
             for text_batch, labels in dataloader(
                     train_features, train_labels, batch_size=batch_size, sequence_length = sequ
                 steps += 1
                 hidden = model.init_hidden(labels.shape[0])

                 # Set Device
```

```
        text_batch, labels = text_batch.to(device), labels.to(device)
        for each in hidden:
            each.to(device)

        # TODO Implement: Train Model
        model.zero_grad()

        output, hidden = model(text_batch, hidden)

        loss = criterion(output.squeeze(), labels)
        loss.backward()

        nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()

        if steps % print_every == 0:
            model.eval()

            # TODO Implement: Print metrics
            validation_losses = []

            for inputs, labels in dataloader(valid_features, valid_labels, batch_size=b
                inputs, labels = inputs.to(device), labels.to(device)

            valid_hidden  = model.init_hidden(labels.shape[0])

            for each in valid_hidden:
                each.to(device)

            output, valid_hidden = model(inputs, valid_hidden)

            validation_loss = criterion(output.squeeze(), labels)
            validation_losses.append(validation_loss.item())

            print(f'Epoch: ({epoch+1}/{epochs}...)',
                  f'Step: {steps}...',
                  f'Validation loss: {np.mean(validation_losses)}')
            model.train()
```

```
Starting epoch 1
Epoch: (1/2...) Step: 100... Validation loss: 1.0287967920303345
Epoch: (1/2...) Step: 200... Validation loss: 0.9034968018531799
Epoch: (1/2...) Step: 300... Validation loss: 0.9425977468490601
Epoch: (1/2...) Step: 400... Validation loss: 0.8036333322525024
Epoch: (1/2...) Step: 500... Validation loss: 0.8390430212020874
Epoch: (1/2...) Step: 600... Validation loss: 0.7831103205680847
Epoch: (1/2...) Step: 700... Validation loss: 0.7563518285751343
Epoch: (1/2...) Step: 800... Validation loss: 0.676030695438385
```

```
Epoch: (1/2...) Step: 900... Validation loss: 0.7376319766044617
Epoch: (1/2...) Step: 1000... Validation loss: 0.7391251921653748
Epoch: (1/2...) Step: 1100... Validation loss: 0.7499340772628784
Epoch: (1/2...) Step: 1200... Validation loss: 0.730837881565094
Epoch: (1/2...) Step: 1300... Validation loss: 0.7465161085128784
Epoch: (1/2...) Step: 1400... Validation loss: 0.7186494469642639
Epoch: (1/2...) Step: 1500... Validation loss: 0.7583974003791809
Epoch: (1/2...) Step: 1600... Validation loss: 0.6628116369247437
Starting epoch 2
Epoch: (2/2...) Step: 100... Validation loss: 0.72899329662323
Epoch: (2/2...) Step: 200... Validation loss: 0.6465509533882141
Epoch: (2/2...) Step: 300... Validation loss: 0.6615562438964844
Epoch: (2/2...) Step: 400... Validation loss: 0.7390872240066528
Epoch: (2/2...) Step: 500... Validation loss: 0.6319352388381958
Epoch: (2/2...) Step: 600... Validation loss: 0.7374707460403442
Epoch: (2/2...) Step: 700... Validation loss: 0.6838297247886658
Epoch: (2/2...) Step: 800... Validation loss: 0.6835922598838806
Epoch: (2/2...) Step: 900... Validation loss: 0.6743893027305603
Epoch: (2/2...) Step: 1000... Validation loss: 0.7536903023719788
Epoch: (2/2...) Step: 1100... Validation loss: 0.7578802704811096
Epoch: (2/2...) Step: 1200... Validation loss: 0.7747513055801392
Epoch: (2/2...) Step: 1300... Validation loss: 0.729171872138977
Epoch: (2/2...) Step: 1400... Validation loss: 0.7179625630378723
Epoch: (2/2...) Step: 1500... Validation loss: 0.678050696849823
Epoch: (2/2...) Step: 1600... Validation loss: 0.6909385919570923
```

1.8 Making Predictions

1.8.1 Prediction

Okay, now that you have a trained model, try it on some new twits and see if it works appropriately. Remember that for any new text, you'll need to preprocess it first before passing it to the network. Implement the `predict` function to generate the prediction vector from a message.

```
In [79]: def predict(text, model, vocab):
             """
             Make a prediction on a single sentence.

             Parameters
             ----------
                 text : The string to make a prediction on.
                 model : The model to use for making the prediction.
                 vocab : Dictionary for word to word ids. The key is the word and the value is t

             Returns
             -------
                 pred : Prediction vector
             """
```

```
            # TODO Implement

            tokens = preprocess(text)

            # Filter non-vocab words
            tokens = [word for word in tokens if word in vocab]
            # Convert words to ids
            tokens = [vocab[word] for word in tokens]

            # Adding a batch dimension
            text_input = torch.tensor(tokens).unsqueeze(1)
            # Get the NN output
            hidden = model.init_hidden(text_input.size(1))
            logps, _ = model.forward(text_input, hidden)
            # Take the exponent of the NN output to get a range of 0 to 1 for each label.
            pred = torch.exp(logps)

            return pred
```

```
In [80]: text = "Google is working on self driving cars, I'm bullish on $goog"
         model.eval()
         model.to("cpu")
         predict(text, model, vocab)
```

```
Out[80]: tensor([[ 0.0009,  0.0226,  0.0141,  0.7300,  0.2325]])
```

1.8.2 Questions: What is the prediction of the model? What is the uncertainty of the prediction?

We have the following scale: very negative, negative, neutral, positive, very positive.

Each score of the list gives the probability that the text falls in each class, in this case the fourth prediction is the greatest (73%). The prediction of the model is that the text has a positive sentiment. The uncertainty is 27%.

Now we have a trained model and we can make predictions. We can use this model to track the sentiments of various stocks by predicting the sentiments of twits as they are coming in. Now we have a stream of twits. For each of those twits, pull out the stocks mentioned in them and keep track of the sentiments. Remember that in the twits, ticker symbols are encoded with a dollar sign as the first character, all caps, and 2-4 letters, like $AAPL. Ideally, you'd want to track the sentiments of the stocks in your universe and use this as a signal in your larger model(s).

1.9 Testing

1.9.1 Load the Data

```
In [81]: with open(os.path.join('..', '..', 'data', 'project_6_stocktwits', 'test_twits.json'),
             test_data = json.load(f)
```

1.9.2 Twit Stream

```
In [82]: def twit_stream():
             for twit in test_data['data']:
                 yield twit

         next(twit_stream())
```

```
Out[82]: {'message_body': '$JWN has moved -1.69% on 10-31. Check out the movement and peers at
          'timestamp': '2018-11-01T00:00:05Z'}
```

Using the `prediction` function, let's apply it to a stream of twits.

```
In [83]: def score_twits(stream, model, vocab, universe):
             """
             Given a stream of twits and a universe of tickers, return sentiment scores for tick
             """
             for twit in stream:

                 # Get the message text
                 text = twit['message_body']
                 symbols = re.findall('\$[A-Z]{2,4}', text)
                 score = predict(text, model, vocab)

                 for symbol in symbols:
                     if symbol in universe:
                         yield {'symbol': symbol, 'score': score, 'timestamp': twit['timestamp']
```

```
In [84]: universe = {'$BBRY', '$AAPL', '$AMZN', '$BABA', '$YHOO', '$LQMT', '$FB', '$GOOG', '$BBB
         score_stream = score_twits(twit_stream(), model, vocab, universe)

         next(score_stream)
```

```
Out[84]: {'symbol': '$AAPL',
          'score': tensor([[ 0.1658,  0.0817,  0.1446,  0.1718,  0.4361]]),
          'timestamp': '2018-11-01T00:00:18Z'}
```

That's it. You have successfully built a model for sentiment analysis!

1.10 Submission

Now that you're done with the project, it's time to submit it. Click the submit button in the bottom right. One of our reviewers will give you feedback on your project with a pass or not passed grade. You can continue to the next section while you wait for feedback.