Code Listings for the Mad Hatter Project

Matey Krastev

May 15, 2023

Introduction

The entirety of the source code included in this document is available on GitHub at: `https://github.com/Rinto-kun/madhatter`.

Mad Hatter documentation is also available at ReadTheDocs: `https://madhatter.readthedocs.io`

Code listings

In what follows, we enumerate all files distributed with this project, for convenience of navigation, we provide an index of all listings below:

Listings

Listing 1: __init__.py

```python
"""
    Mad Hatter.
"""

from .benchmark import *
from .metrics import *
from .models import *

import os.path

if not os.path.exists(f"{__file__}/static"):
    from .loaders import load_concreteness, load_freq, load_imageability

    load_freq()
    load_concreteness()
    load_imageability()
```

Listing 2: __main__.py

```python
import argparse


def main():
    parser = argparse.ArgumentParser(
        prog='madhatter',
        description='A command-line utility for generating book project reports.'
            ,
    )

    parser.add_argument('filename', help="text file to parse")
    parser.add_argument('-p', '--postag', action="store_true",
                        help='whether to return a POS tag distribution over the
                            whole text')
    parser.add_argument('-u', '--usellm', action="store_true",
                        help='whether to run GPU-intensive LLMs for additional
                            characteristics')
    parser.add_argument(
        '-m', '--maxtokens', help="maximum number of predicted tokens for the
            heavyweight metrics. Tokens start from the beginning of text, -1 to
            read until the end", default=1000, type=int
    )
    parser.add_argument(
        '-c', '--context', help='context length for sliding window predictions as
             part of heavyweight metrics', default=10, type=int
    )
    parser.add_argument(
        '-t', '--title', help='optional title to use for the report project.'
    )
    parser.add_argument(
        '-d', '--tagset', help='tagset to use', default="universal"
    )

    args = parser.parse_args()

    from .benchmark import CreativityBenchmark

    with open(args.filename) as f:
        text = f.read()

    bench = CreativityBenchmark(text, args.title, args.tagset)

    print(bench.report(False, args.postag,
          args.usellm, n=args.maxtokens, k=args.context))


main()
```

Listing 3: Benchmark

```
"""Main Class for the Application Logic"""

# pylint: disable=missing-function-docstring, invalid-name
from itertools import chain
from time import time
from typing import Any, Callable, Generator, Iterable, NamedTuple, Optional,
   Tuple

import matplotlib.pyplot as plt
import nltk
import numpy as np
import pandas as pd
import seaborn as sns
from nltk.corpus import wordnet as wn
from nltk.stem import WordNetLemmatizer
from scipy.interpolate import make_interp_spline
from .models import default_model, sent_predictions, sliding_window_preds_tagged
from .metrics import _ratings, predictability, surprisal
from .utils import (get_concreteness_df, get_freq_df, get_imageability_df, mean,
    slope_coefficient, stopwords)

sns.set_theme()

TAG_TO_WN = {
    "NOUN": wn.NOUN,
    "VERB": wn.VERB,
    "ADJ": wn.ADJ,
    "ADV": wn.ADV
}

TAGS_OF_INTEREST = {'NOUN', 'VERB', 'ADJ'}  # ignore 'ADV'

class BookReport(NamedTuple):
    """Report object
    """
    title: str
    nwords: Optional[int] = None
    mean_wl: Optional[float] = None
    mean_sl: Optional[float] = None
    mean_tokenspersent: Optional[float] = None
    prop_contentwords: Optional[float] = None
    mean_conc: Optional[float] = None
    mean_img: Optional[float] = None
    mean_freq: Optional[float] = None
    prop_pos: Optional[dict] = None
    surprisal: Iterable | None = None
    predictability: Iterable | None = None

    # for debugging
    def __str__(self):
        newline = "\n\t"
        return f"BookReport({newline.join(f'{_[0]}={_[1]}' for _ in (self._asdict
            ().items()))})" # pylint: disable=no-member

class CreativityBenchmark:
    """
        This class is used to benchmark the creativity of a text.
    """

    plots_folder = 'plots/'
    # TODO: change the tagset to be benchmark-tied
    tags = {'.', 'ADJ', 'ADP', 'ADV', 'CONJ', 'DET', 'NOUN', 'NUM', 'PRON', 'PRT'
        , 'VERB', 'X'}
    tags_of_interest = set(['NOUN', 'VERB', 'ADJ'])  # ignore 'ADV'
    tag_to_embed = {tag: i for i, tag in enumerate(tags)}
    embed_to_tag = {i: tag for i, tag in enumerate(tags)}

    def __init__(self, raw_text: str, title: str = "unknown", tagset: str = '
        universal'):
        self.raw_text = raw_text
        self.words = nltk.word_tokenize(raw_text, preserve_line=True)
        self.sents = nltk.sent_tokenize(self.raw_text)
        self.tokenized_sents = [
            nltk.word_tokenize(sent) for sent in self.sents]

        self.tagset = tagset
        self.tagged_sents = nltk.pos_tag_sents(
            self.tokenized_sents, tagset=self.tagset)
        # self.sents = [nltk.word_tokenize(sent) for sent in self.sents]

        # Initialize a list to hold the POS tag counts for each sentence
        self.postag_counts: list[nltk.FreqDist] = []
        self.title = title

        self._tagged_words: list[tuple[str,str]] | None = None

    def ngrams(self, n, **kwargs):
        """Returns ngrams for the text."""
        return nltk.ngrams(self.raw_text, n, kwargs)  # type: ignore # pylint:
            disable=too-many-function-args

    def sent_postag_counts(self, tagset: str = "universal") -> list[nltk.FreqDist
       ]:
        """Returns sentence-level counts of POS tags for each sentence in the
            text. """
        if self.postag_counts and self.tagset == tagset:
            return self.postag_counts
        else:
            self.tagset = tagset
            # Collect POS data for each sentence
            for sentence in self.tagged_sents:
                # Initialize a counter for the POS tags on the sentence level
                lib = nltk.FreqDist()
                for _, token in sentence:
                    lib[token] += 1

                self.postag_counts.append(lib)

            return self.postag_counts

    @property
    def tagged_words(self):
        if self._tagged_words is not None:
            return self._tagged_words

        self._tagged_words = list(chain.from_iterable(self.tagged_sents))
        return self._tagged_words

    def book_postag_counts(self, tagset: Optional[str] = None) -> nltk.FreqDist:
        """Get a counter object for the Parts of Speech in the whole book."""

        if not tagset:
            tagset = self.tagset
        # Opt to use this instead for consistency.
        book_total_postags = nltk.FreqDist()
        for l in self.sent_postag_counts(tagset=tagset):
            book_total_postags += l
        return book_total_postags

    def num_tokens_per_sentence(self) -> Generator[int, None, None]:
        """Returns a generator for the number of tokens in each sentence."""
        return (len(sentence) for sentence in self.tokenized_sents)

    def total_tokens_per_sentence(self) -> int:
        return sum(self.num_tokens_per_sentence())

    def avg_tokens_per_sentence(self) -> float:
        return sum(self.num_tokens_per_sentence())/len(self.sents)
```

```
def postag_graph(self):
    # Potentially consider color schemes for nounds, adjectives, etc., not
       just a random one
    book_total_postags = self.book_postag_counts(tagset='universal')

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(20, 8))
    sns.barplot(x=list(book_total_postags.keys()), y=list(
        book_total_postags.values()), label=self.title, ax=ax1)
    ax1.set_title(f"POS Tag Counts for {self.title}")
    ax1.set_ylabel("Count")
    ax1.set_ylim(bottom=30)

    # Set counts to appear with the K suffix
    ax1.yaxis.set_major_formatter(plt.FuncFormatter(  # type: ignore
        lambda x, loc: f"{int(x/1000):,}K"))
    ax1.tick_params(axis='x', labelrotation=90)

    num_tokens_per_sentence = list(self.num_tokens_per_sentence())
    # x = np.arange(0, num_tokens_per_sentence.shape[0])
    # spline = make_interp_spline(x, num_tokens_per_sentence, 3, bc_type='
       natural')
    ax2.set(title="Distribution of tokens per sentence", xlabel="Sentence #",
            ylabel="(Any) token count", ylim=(10, 300), xlim=(-50, len(
               num_tokens_per_sentence) + 100))
    ax2.plot(num_tokens_per_sentence)

    fig.subplots_adjust(hspace=0.8)

def plot_postag_distribution(self, fig=None, ax=None, **kwargs) -> Tuple[Any,
    Any]:
    '''
    Plots a stackplot of the POS tag counts for each sentence in a book.
    Note: works best with a Pandas dataframe with the columns as the POS tags
        and the rows as the sentences.

    TODO: Optionally, set more options for modifying the figure, e.g.
       linewidth, color palette, etc.
    '''

    df = pd.DataFrame(self.sent_postag_counts(tagset='universal'))
    # Fill in any missing values with 0
    df.fillna(0, inplace=True)
    # Divide each row by the sum of the row to get proportions
    df = df.div(df.sum(axis=1), axis=0)

    if fig is None or ax is None:
        fig, ax = plt.subplots(**kwargs)
    xnew = np.linspace(0, df.shape[0], 100)

    graphs = []
    # For each PosTag, create a smoothed line
    for label in df.columns:
        spl = make_interp_spline(
            list(df.index), df[label], bc_type='natural')  # BSpline object
        power_smooth = spl(xnew)

        graphs.append(power_smooth)

    ax.stackplot(xnew, *graphs, labels=df.columns, linewidth=0.1,
                 colors=sns.color_palette("deep", n_colors=df.shape[1],
                    as_cmap=True))

    ax.set(xbound=(0, df.shape[0]), ylim=(0, 1), title=f'Parts of Speech in {
       self.title}',
           xlabel='Sentence #', ylabel='Proportion of sentence')
    ax.xaxis.set_major_formatter(plt.FuncFormatter(  # type: ignore
        lambda x, loc: f"{x/df.shape[0]:.0%}"))

    ax.legend()
    return fig, ax

def plot_transition_matrix(self):
    """
    Plots a transition matrix for the POS tags in a book.
    """
    tagged_words = nltk.pos_tag(self.words, tagset='universal')

    counter = np.zeros((len(self.tags), len(self.tags)), dtype=np.int32)
    for pair_1, pair_2 in zip(tagged_words, tagged_words[1:]):
        counter[self.tag_to_embed[pair_1[1]],
                self.tag_to_embed[pair_2[1]]] += 1
    counter = (counter - counter.mean()) / counter.std()

    plt.figure(figsize=(20, 20))
    plt.imshow(counter, cmap="Blues")
    for i in range(counter.shape[0]):
        for j in range(counter.shape[1]):
            plt.text(j, i, f"{self.embed_to_tag[i]}/{self.embed_to_tag[j]}",
               ha="center",
                     va="bottom", color="gray", fontdict={'size': 14})
            plt.text(j, i, f"{counter[i, j]:.3f}", ha="center",
                     va="top", color="gray", fontdict={'size': 18})
    plt.title(
        f"POS Tag Transition Matrix for {self.title} with {len(tagged_words)}
            words")
    plt.axis('off')

# def get_synsets(word):

def avg_word_length(self):
    return sum(len(word) for word in self.words)/len(self.words)

def avg_sentence_length(self):
    return sum(len(sentence) for sentence in self.sents)/len(self.sents)

def content_words(self):
    return (word for word in self.words if word not in stopwords)

def content_word_sentlevel(self):
    """Discards stopwords

    Returns:
        list[list[str]]: A list of sentences containing the word tokens.
    """
    return [[word for word in sent if word not in stopwords] for sent in self
       .tokenized_sents]

def ncontent_word_sentlevel(self):
    return [len(sent) for sent in self.content_word_sentlevel()]
```

```
def calculate_sent_slopes(self, model, tokenizer, n) -> list[list[float]]:
    # Returns slopes for the __words__ of the first ‘n‘ sentences of the ‘
       sents‘ list of sentences.
    res = []
    for sent in self.tokenized_sents[:n]:
        results = sent_predictions(sent, self, model, tokenizer, False)

        res.append(
            [slope_coefficient(
                np.arange(len(result.probs)),
                np.array(result.probs))
             for result in results
             if len(result) > 0]
        )

    return res

@property
def model(self):
    return self.model

@property
def word2vec_model(self):
    return self.word2vec_model

def calculate_sim_scores(self, model, tokenizer, sim_function: Callable,
    max_sents=-1):
    similarity_scores = []
    for sent in self.tokenized_sents[:max_sents]:

        preds = sent_predictions(
            sent, model, tokenizer, True, k=10)

        average_position_of_correct_prediction = 0
        # number of predictions which do not include the true value in the
            topmost k results
        missed_predictions = 0
        # note that word here is a tuple of the word and its POS tag
        i = 0
        for pred in preds:
            try:
                # print(word[0], predlist)
                average_position_of_correct_prediction += pred.suggestions.
                    index(
                     pred.word)
                i += 1
            except ValueError:
                missed_predictions += 1

        # Avoid division by zero error
        if i == 0:
            average_position_of_correct_prediction = 0
        else:
            average_position_of_correct_prediction /= i
        similarity_scores.append(
            (average_position_of_correct_prediction, missed_predictions))
        break
        #     for item in predlist:
        # similarity_scores.append(
        #     [[sim_function(word[0], pred) for pred in predlist] for word,
            predlist in predictions.items()]
        # )

    return similarity_scores

def predictability(self, n: int, model, tokenizer):
    return [item for sublist in self.calculate_sent_slopes(model, tokenizer,
       n) for item in sublist]

def plot_predictability(self, n, model, tokenizer):
    plotting_list = - np.array(self.predictability(n, model, tokenizer))
    plt.figure(figsize=(20, 8))
    plt.plot(plotting_list)
    plt.title(
        f"Aggregate predictability of content words within the context of
            first {n} sentences (Mean: {plotting_list.mean():.3f})")
    plt.xlabel('Word #')
    plt.ylabel('Predictability')

    plt.show()

# def sim_func(word: str, pred: str) -> float | None:
#     """Arbitrary function to use when calculating vector similarity between
     the embeddings of two words. Serves as an example.

#     Parameters
#     ----------
#     word : str
#         Normally, the original (true) value.
#     pred : str
#         Normally, the predicted value.

#     Returns
#     -------
#     Optional[float]
#         Can return a float or None.
#     """
#     try:
#         return word2vec_model.similarity(word, pred)
#     except:
#         pass
def sent_lemmas(self) -> list[list[str]]:
    lemmatizer = WordNetLemmatizer()
    return [
        [lemmatizer.lemmatize(word, TAG_TO_WN[tag]) for word, tag in sent if
            tag in TAG_TO_WN] for sent in self.tagged_sents
    ]

def lemmas(self) -> list[str]:
    lemmatizer = WordNetLemmatizer()
    return [
        lemmatizer.lemmatize(word, TAG_TO_WN[tag]) for sent in self.
            tagged_sents for word, tag in sent if tag in TAG_TO_WN
    ]

def frequency_ratings(self, lemmas: Optional[list[str]] = None) -> list[
    Optional[float]]:
    return _ratings(self.lemmas(), get_freq_df("dict")) if lemmas is None
       else _ratings(lemmas, get_freq_df("dict"))

def concreteness_ratings(self, lemmas: Optional[list[str]] = None) -> list[
    Optional[float]]:
    return _ratings(self.lemmas(), get_concreteness_df("dict")) if lemmas is
       None else _ratings(lemmas, get_concreteness_df("dict"))

def imageability_ratings(self, lemmas: Optional[list[str]] = None) -> list[
    Optional[float]]:
    return _ratings(self.lemmas(), get_imageability_df("dict")) if lemmas is
       None else _ratings(lemmas, get_imageability_df("dict"))


def report(self, print_time=False, include_pos=True, include_llm=False, n =
    1000, model = None, tokenizer = None, k = 10, word2vec_model=None) ->
    BookReport:
    """
        Generates a report for the text.
    """
    lemmas = self.lemmas()

    postag_dist = {}

    time_now = time() if print_time is True else 0.0

    ncontent_words = self.ncontent_word_sentlevel()
    # avg_num_content_words = mean(ncontent_words)
    ratio_content_words = sum(1 for _ in ncontent_words) / len(self.words)

    conc = [_ for _ in self.concreteness_ratings(lemmas) if _]
    conc_num = mean(conc)

    image = [_ for _ in self.imageability_ratings(lemmas) if _]
    image_num = mean(image)

    freq = [_ for _ in self.frequency_ratings(lemmas) if _]
    freq_num = mean(freq)

    if include_pos:

        # The postagging takes a while
        postag_counts = self.book_postag_counts()
        total = sum(i for i in postag_counts.values())
        postag_dist = {tag: val/total for tag,
                       val in postag_counts.items() if tag in self.
                          tags_of_interest}

    _surprisal, _predictability = None, None
    if include_llm is True:
        if model is None or tokenizer is None:
            model, tokenizer = default_model()
        preds = sliding_window_preds_tagged(self.tagged_words[:n], model,
            tokenizer, return_tokens=True, k=k, tags_of_interest=self.
            tags_of_interest, stopwords=stopwords)

        _surprisal = surprisal(preds, word2vec_model)

        _predictability = predictability(preds)


    result = BookReport(self.title, len(self.words), self.avg_word_length(),
       self.avg_sentence_length(
    ), self.avg_tokens_per_sentence(), ratio_content_words, conc_num,
       image_num, freq_num, postag_dist, _surprisal, _predictability)
    if print_time is True:
        print(f"Report took ~{time() - time_now:.3f}s")

    return result


def plot_report(self, global_dist: BookReport, categories: list[str] = ["mean_wl", "mean_sl", "prop_contentwords", "mean_conc", "mean_img", "mean_freq"], **report_args):

    report = self.report(**report_args)

    # number of variable
    N = len(categories)

    dfnp = np.array([getattr(report, cat) for cat in categories])
    norm = np.array([getattr(global_dist, cat) for cat in categories])

    dfnorm = dfnp / norm

    # But we need to repeat the first value to close the circular graph:
    values = dfnorm
    values = np.append(values, values[0])

    # What will be the angle of each axis in the plot? (we divide the plot / number of variable)
    angles = np.arange(N) * 2 * np.pi / N
    angles = np.append(angles, angles[0])

    fig, ax = plt.subplots(subplot_kw={'projection': 'polar'}, dpi=200)

    # Draw one axe per variable + add labels
    ax.set_xticks(angles[:-1], categories, color='grey', size=8)
    # Draw ylabels
    ax.set(rlabel_position=0)
    yticks = np.linspace(0, 1, 5)
    ax.set_yticks(yticks, yticks, color="grey", size=7)
    ax.set_ylim(0, 1)

    # Plot data
    ax.plot(angles, values, linewidth=1, linestyle='solid')

    # Fill area
    ax.fill(angles, values, 'b', alpha=0.1)

    return fig, ax
```

Listing 4: Models

```
"""models.py
Base file for LLM operations on text.
"""

# Initialize models

import torch
from nltk import pos_tag, word_tokenize
from typing import Any, Literal, NamedTuple


class Prediction(NamedTuple):
    ”””Prediction class. Contains:
    ‘‘‘
    word: str
    original_tag: str
    suggestions: list[str]
    probs: list[float]
    ‘‘‘
    ”””
    word: str
    original_tag: str
    suggestions: list[str]
    probs: list[float]

    def __bool__(self):
        return len(self.suggestions) == len(self.probs)


def predict_tokens(sent: str, masked_word: str, model, tokenizer, return_tokens:
   bool = True, max_preds: int = 20) -> tuple[list[float], list[str]]:
    ”””
    Predict the top k tokens that could replace the masked word in the sentence.

    Returns a list of tuples of the form (token, likelihood, similarity) where
        similarity is the cosine similarity of the given words in a word2vec model
        .

    Parameters
    ----------
    sent: str
        The sentence to predict tokens for.
    masked_word: str
        The word to predict tokens for. Note that this word must be in the
            sentence.
    model
        Must be a masked language model that takes in a sentence and returns a
            tensor of logits for each token
        in the sentence. Default assumes a pretrained BERT model from the
            HuggingFace ‘transformers‘ library.
    word2vec_model
        Must be a word2vec model that takes in a word and returns a vector
            representation of the word.
        Default is ‘gensim.models.keyedvectors.KeyedVectors‘ loaded from the ‘
            word2vec_sample‘ model
        from the ‘nltk_data‘ package.
    k: int
        The number of tokens to return.

    Returns
    -------
    List of tuples the form (token, likelihood)

    token: str
        The predicted token.
    likelihood: float
        The likelihood of the token being the masked word.
    ”””
    if masked_word not in sent:
        raise ValueError(f"{masked_word} not in {sent}")
    masked_sent = sent.replace(masked_word, "[MASK]")

    inputs = tokenizer(masked_sent, return_tensors="pt")
    with torch.no_grad():
        logits = model(**inputs).logits

    # retrieve index of [MASK]
    mask_token_index = (inputs.input_ids == tokenizer.mask_token_id)[
        0].nonzero(as_tuple=True)[0]

    vals, predicted_token_ids = torch.topk(  # pylint: disable=no-member
        logits[0, mask_token_index], max_preds, dim=-1)

    ret = []
    ret_tokens = []
    for i, predicted_token_id in enumerate(predicted_token_ids[0]):
        # if the actual tokens are needed, return those as well
        if return_tokens is True:
            word = tokenizer.decode(predicted_token_id)

            # If word is a subword, combine it with the previous word
            word = word if not word.startswith("##") else masked_word+word[2:]

            ret_tokens.append(word)

        ret.append(vals[0, i].item())

    return ret, ret_tokens


def sent_predictions(sent: str | list[str], model: Any, tokenizer: Any,
   return_tokens: Literal[True, False] = False, k: int = 20, stopwords: set |
   None = None, tags_of_interest: set | None = None) -> list[Prediction]:
    """Returns predictions for content words in a given sentence. If
       return_tokens is true,
    returns a key-value pair dictionary where the key is the used word, and the
       value is a list of suggested tokens,
    corresponding to the likekihoods in the first list.
    """
    if isinstance(sent, str):
        tokens = word_tokenize(sent.lower())
    elif isinstance(sent, list):
        tokens = [token.lower() for token in sent]
        sent = " ".join(tokens)
    else:
        raise TypeError()
    words = pos_tag(tokens, tagset='universal')

    results = []

    if stopwords is None:
        stopwords = set()

    if tags_of_interest is None:
        tags_of_interest = set()

    # loop over the words of the sentence
    for word, tag in words:
        # Early stopping
        if word in stopwords or tag not in tags_of_interest:
            continue

        probs, tokens = predict_tokens(
                sent, word, model, tokenizer, return_tokens=return_tokens,
                    max_preds=k)

        results.append(Prediction(word, tag, tokens, probs))

    return results


def sliding_window_preds_tagged(words: list[tuple[str, str]], model: Any,
    tokenizer: Any, return_tokens: Literal[True, False] = False, k: int = 20,
    max_preds: int = 10, stopwords: set | None = None, tags_of_interest: set |
    None = None) -> list[Prediction]:
    """
        Note: must be used in conjunction with a list of tuples with already
            tagged words.
    """
    if not isinstance(words, list):
        raise ValueError(
            "Incorrect values passed for ‘words‘, expected a list of tuples")

    if stopwords is None:
        stopwords = set()

    if tags_of_interest is None:
        tags_of_interest = set()

    results = []

    # loop over the words of the sentence
    for i, (word, tag) in enumerate(words[k:-k], start=k):
        # Early stopping
        if word in stopwords or tag not in tags_of_interest:
            continue

        sent_tuples = words[i-k:i+k]
        sent = " ".join(_[0] for _ in sent_tuples)

        probs, tokens = predict_tokens(
                sent, word, model, tokenizer, return_tokens=return_tokens,
                    max_preds=k)

        results.append(Prediction(word, tag, tokens, probs))

    return results


def sliding_window_preds(_words: list[str], model: Any, tokenizer: Any,
    return_tokens: Literal[True, False] = False, k: int = 20, max_preds: int = 10,
     stopwords: set | None = None, tags_of_interest: set | None = None) -> list[
    Prediction]:
    """
        Returns a list of predictions given the sliding window for context on the
             model predictions.
    """
    if not isinstance(_words, list):
        raise ValueError(
            "Incorrect values passed for ‘words‘, expected a list of strings")

    if len(_words) < k:
        raise ValueError(
            f'The given window ({_words=}) contains less tokens than the
```

```
               requested sliding window({k=}); ')

    words = pos_tag(_words, tagset='universal')

    if stopwords is None:
        stopwords = set()

    if tags_of_interest is None:
        tags_of_interest = set()

    results = []

    # loop over the words of the sentence
    for i, (word, tag) in enumerate(words[k:-k], start=k):
        # Early stopping
        if word in stopwords or tag not in tags_of_interest:
            continue

        sent_tuples = words[i-k:i+k]
        sent = " ".join(_[0] for _ in sent_tuples)

        probs, tokens = predict_tokens(
                sent, word, model, tokenizer, return_tokens=return_tokens,
                   max_preds=k)

        results.append(Prediction(word, tag, tokens, probs))


    return results


def default_model(model_name="bert-base-uncased"):
    from transformers import AutoTokenizer, BertForMaskedLM

    return BertForMaskedLM.from_pretrained(model_name), AutoTokenizer.
       from_pretrained(model_name)


def default_word2vec():
    import gensim
    from nltk.data import find

    return gensim.models.KeyedVectors.load_word2vec_format(
        str(find('models/word2vec_sample/pruned.word2vec.txt')), binary=False)
```

Listing 5: Metrics

```
import numpy as np
import numpy.typing as ntp
from typing import Any, Literal, Optional
from nltk.corpus import wordnet as wn
from nltk.corpus import wordnet_ic as ic
import pandas as pd

from .models import Prediction
from .utils import get_freq_df, mean, stopwords


def imageability(data: str | list[str], imageability_df: pd.DataFrame) ->
   Optional[float] | list[Optional[float]]:
    """Returns the mean imageability rating for a given word or list of words,
        according to the table of ~40,000 words and word definitions, as defined
        by Brysbaert et al (2013)."""
    # TODO: Possibly look at amortized values given standard deviations

    # Fastest way for lookups so far.
    dictionary = dict(
        zip(imageability_df["item"], imageability_df["rating"]))

    return _ratings(data, dictionary)

def concreteness(data: str | list[str], concreteness_df: pd.DataFrame) -> float |
     None | list[float | None]:
    """Returns the mean concreteness rating for a given word or list of words,
        according to the table of ~40,000 words and word definitions, as defined
        by Brysbaert et al (2013)."""
    # TODO: Possibly look at amortized values given standard deviations

    # Fastest way for lookups so far.
    conc = dict(
        zip(concreteness_df["Word"], concreteness_df["Conc.M"]))

    return _ratings(data, conc)

def frequency_ratings(data):
    """Returns log10 frequency for lemmatized words"""
    df_dict = get_freq_df("dict")  # 6652 entries

    return _ratings(data, df_dict)  # type: ignore


def _ratings(data, func: dict):
    """j"""

    if isinstance(data, str):
        return func.get(data.lower(), None)
    if isinstance(data, list):
        # type: ignore
        return [func.get(w.lower(), None) for w in data if w not in stopwords]

    raise TypeError(
        f"Inappropriate argument type for `word`. Expected `list` or `str`, but
           got {type(data)}")



def _word2vec_similarity(first: str, second: str, word2vecmodel) -> float | None:
    try:
        return word2vecmodel.similarity(first.lower(), second.lower())
    except:
        return None


def word2vec_similarity(preds: list[Prediction], word2vecmodel) -> list[float]:
    # Return the mean similarity between tokens
    return [mean(list(x for x in list(_word2vec_similarity(pred.word, sug,
        word2vecmodel) for sug in pred.suggestions) if x)) for pred in preds]


def _lin_similarity(first: str, second: str, pos: Literal['n', 'a', 's', 'r', 'v'
   ], ic_dict) -> float | None:
    """Generates similarity using Lin Similarity via the information content
```

```
        present in ic_dict. For words with multiple definitions, we skip disambiguation and select the most common definition.

    Parameters
    ----------
    first : str
        The first word
    second : str
        Second word
    pos : one of 'n', 'a', 's', 'r', 'v'
        The part of speech of the given words
    ic_dict : dict
        Information content dictionary, must contain both words.

    Returns
    -------
    float | None
        A float result if both words can be found in the ic_dict.
    """

    try:
        return wn.synset(f"{first.lower()}.{pos}.1").lin_similarity(wn.synset(f"{second.lower()}.{pos}.1"), ic_dict)
    except:
        return None


def wordnet_lin_similarity(preds: list[Prediction], ic_dict, tag_to_wn: dict):

    # Return the mean similarity between tokens
    return [mean(list(x for x in list(_lin_similarity(pred.word, sug, tag_to_wn[pred.original_tag], ic_dict) for sug in pred.suggestions) if x)) for pred in preds]



def _wup_similarity(first: str, second: str, pos: Literal['n', 'a', 's', 'r', 'v']) -> float | None:
    try:
        return wn.synset(f"{first.lower()}.{pos}.1").wup_similarity(wn.synset(f"{second.lower()}.{pos}.1"))
    except:
        return None



def wordnet_wup_similarity(preds: list[Prediction], tag_to_wn: dict):
    """Returns the mean Wu-Palmer path similarity for the tokens."""
    return [mean(list(x for x in list(_wup_similarity(pred.word, sug, tag_to_wn[pred.original_tag]) for sug in pred.suggestions) if x)) for pred in preds]


def predictability(preds: list[Prediction]) -> ntp.NDArray:
    """Returns an array calculating the predictability metric, defined as a mean over the gradient of each prediction.
    Parameters
    ----------
    preds : list[Prediction]
        A list of predictions to be used

    Returns
    -------
    NDArray
        A numpy array for the predictability values of each prediction.
    """
    return -np.gradient(np.array(list(pred.probs for pred in preds)), axis=1).
        mean(axis=1)


def surprisal(preds: list[Prediction], word2vec_model: Any | None = None) -> ntp.
    NDArray:

    if word2vec_model is None:

        import gensim
        from nltk.data import find

        # TODO: Implement non-hardcoded with NLTK loading.
        word2vec_model = gensim.models.KeyedVectors.load_word2vec_format(
            str(find('models/word2vec_sample/pruned.word2vec.txt')), binary=False
                )

    return np.array(word2vec_similarity(preds, word2vec_model))
```

Listing 6: Utilities

```
import pkgutil
from io import BytesIO
from typing import Literal, Sequence

import numpy as np
import numpy.typing as ntp
import pandas as pd

# from .loaders import load_ concreteness, load_imageability


def mean(items: Sequence) -> float:
    return 0 if len(items) == 0 else sum(items)/len(items)


def cross_softmax(one: ntp.NDArray, two: ntp.NDArray, temp1=0.5, temp2=0.5):
    # return (torch.softmax(torch.from_numpy(results[1][:,0]), dim=0) @ torch.
        softmax(torch.from_numpy(results[1][:,1]), dim=0)).item()
    exps = np.exp(one*temp1)
    exps /= exps.sum()
    exps2 = np.exp(two*temp2)
    exps2 /= exps2.sum()
    return exps @ exps2


def slope_coefficient(one: ntp.NDArray, two: ntp.NDArray) -> float:
    """Returns the coefficient of the slope"""
    # Using the integrated function
    # return np.tanh(np.polyfit(x,y,1)[0])
    # Manually implementing slope equation
    return ((one*two).mean(axis=0) - one.mean()*two.mean(axis=0)) / ((one**2).
        mean() - (one.mean())**2)


def get_concreteness_df(_format: Literal['df', 'dict'] = "df") -> pd.DataFrame |
    dict:
```

```
    # load_concreteness()
    dataframe = pd.read_csv(BytesIO(pkgutil.get_data(
        __package__, 'static/concreteness/concreteness.csv')), sep="\t")  # type:
            ignore
    # concreteness_df = concreteness_df.set_index("Word").sort_index()

    return dataframe if _format == "df" else dict(
        zip(dataframe["Word"], dataframe["Conc.M"]))


def get_imageability_df(_format="df") -> pd.DataFrame | dict:
    """Returns a table of the imageability of ~40,000 words and word definitions,
        as defined by Brysbaert et al (2013)."""

    # load_imageability()
    # Dicts are the fastest way to make string accesses
    dataframe = pd.read_csv(BytesIO(pkgutil.get_data(
        __package__, "static/imageability/cortese2004norms.csv")), header=9)  #
            type: ignore
    return dataframe if _format == "df" else dict(
        zip(dataframe["item"], dataframe["rating"]))


def get_freq_df(_format) -> pd.DataFrame | dict:
    '''
    Key:

    Word = Word type (headword followed by any variant forms) - see pp.4-5

    PoS  = Part of speech (grammatical word class - see pp. 12-13)

    Freq = Rounded frequency per million word tokens (down to a minimum of 10
        occurrences of a lemma per million)- see pp. 5

    Ra   = Range: number of sectors of the corpus (out of a maximum of 100) in
        which the word occurs

    Disp = Dispersion value (Juilland's D) from a minimum of 0.00 to a maximum of
        1.00.
    '''

    df_freq = pd.read_csv(BytesIO(pkgutil.get_data(
        __package__, "static/frequency/frequency.csv")), encoding='unicode_escape
            ', sep="\t")  # type: ignore

    # drop unnamed column one cuz trash source,
    # also remove the column storing word variants
    df_freq = df_freq.drop([df_freq.columns[0], df_freq.columns[3]], axis=1)

    # clean the data
    df_freq = df_freq.convert_dtypes()
    df_freq["Freq"] = pd.to_numeric(df_freq["Freq"], errors="coerce")
    df_freq["Ra"] = pd.to_numeric(df_freq["Ra"], errors="coerce")
    df_freq["Disp"] = pd.to_numeric(df_freq["Disp"], errors="coerce")
    df_freq = df_freq.dropna()

    # filter out the word variants
    df_freq = df_freq.loc[df_freq["Word"] != '@']

    # replace the PoS tags with the ones we are using
    def replace_df(_df, column, replacements) -> pd.DataFrame:
        for src, tar in replacements:
            _df.loc[_df[column].str.contains(src), column] = tar
        return _df

    replacements = [("No", "NOUN"), ("Adv", "ADV"),
                    ("Adj", "ADJ"), ("Verb", "VERB")]
    df_freq = replace_df(df_freq, "PoS", replacements)

    # Scale logarithmically for a better result
    df_freq["Freq"] = np.log10(df_freq["Freq"])

    if _format == "df":
        return df_freq

    # # set the index to the "Word" column so lookups are faster
    # df = df.set_index('Word')

    # I don't particularly care enough for disambiguating their PoS tags :skull:,
         so might as well aggregate the columns and make it even faster.
    # group everything together because i literally cant bother with pos tag
        lookups on big scales
    df_freq = df_freq.groupby('Word').sum(numeric_only=True)

    # df_freq_dict = dict(zip(((x,y) for x, y in zip(df_freq.index, df_freq["PoS
        "]) if y in TAGS_OF_INTEREST), df_freq["Freq"])) # 5600 entries

    return dict(zip(df_freq.index, df_freq["Freq"]))  # 5900 entries


stopwords = {'of', 'been', "hadn't", "isn't", 'i', 'this', 'these', 'were', 'the'
    , 'and', 'by', 'don', 'm', 'o', "wasn't", 'we', 'all', 'same', 'not', 'weren',
     'at', 'those', 'few', 'shan', 'a', 'through', 'ain', 'its', 'how', "that'll",
     'ours', 'you', 'here', 'nor', "weren't", 'myself', 'aren', 'why', "didn't", '
    having', 'for', 'so', 'she', "mightn't", 'in', 'haven', 't', 'being', '
    yourself', 'an', 'to', 'didn', 'between', 'them', "couldn't", "mustn't", '
    itself', 'is', 'only', "aren't", 'very', "you'll", 'had', 'into', 'if', 'their
    ', 'mustn', 'off', 'what', 'd', 'as', 'ourselves', 'that', 'hasn', 'each', 'me
    ', 'below', "haven't", 'wouldn', 'shouldn', 'there', 'your', 'or', 'such', '
    because', 'during', 'yourselves', 'other', 'hadn',
                "should've", 'own', 'mightn', 'our', 'y', 'after', 'on', "doesn't",
                    'ma', 'more', 'again', 'out', 'when', "you've", 'above', 'whom',
                    'under', 'have', 'll', 're', 've', 'isn', 'too', 'won', 'which',
                    'until', "you're", 'up', "hasn't", 'about', 'while', 'needn', '
                    wasn', 'doesn', 'once', 'he', 'my', 'they', 'him', 'does', 'her',
                     'most', 'am', 'further', 'then', 'some', 'herself', 'than', '
                    yours', 'over', 'down', 's', 'both', 'themselves', "won't", "shan
                    't", 'can', "wouldn't", 'has', 'hers', 'did', 'against', 'be', "
                    shouldn't", 'doing', "don't", 'will', 'his', 'no', 'should', "you
                    'd", 'theirs', 'couldn', 'do', 'any', "it's", 'who', 'with', '
                    from', 'was', 'himself', 'it', 'just', 'now', 'but', 'before', "
                    needn't", 'where', 'are', "she's"}.union("!\"#$%&\'()*+,-./:;<=>?
                    @[\\]^_`{|}~")
```

Listing 7: Loaders

```
"""Provides the data loaders for the datasets that may be used in the pipelines.
    """

import tarfile
import zipfile
```

~~~python
from io import BytesIO
from pathlib import Path

from requests import get
from tqdm import tqdm


def ds_cloze(path="./data") -> dict[str, Path]:
    """Returns a dataset object for interacting with the cloze test dataset as
        extracted by XX

    Parameters
    ----------
    path : str, optional
        Default path for storing the files, by default "./data/"

    Returns
    -------
    dict[str,Path]
        A dictionary object with the following structure:
        ```
        - split: path
        ```
        Where source is one of `[test, train, val]` and path is a Path object
            pointing to the csv file of the dataset.

    Example Usage
    -------
    ```
    import pandas as pd

    ds = ds_cloze()
    df = pd.read_csv(ds["train"])
    df.head()
    ```
    """
    clozepath = Path(path) / "cloze/"

    trainpath = clozepath / "cloze_train.csv"
    testpath = clozepath / "cloze_test.csv"
    valpath = clozepath / "cloze_val.csv"
    if not clozepath.exists():
        clozepath.mkdir(exist_ok=True)
        trainpath.write_bytes(get("https://goo.gl/0OYkPK", timeout=5).content)

        testpath.write_bytes(get("https://goo.gl/BcTtB4", timeout=5).content)

        valpath.write_bytes(get("https://goo.gl/XWjas1", timeout=5).content)

    return {"test": testpath, "train": trainpath, "val": valpath}


def tiny_shakespeare(path="./data/"):
    """p """
    tiny_shakespeare_path = Path(
        path) / "tiny_shakespeare" / "tiny_shakespeare.txt"
    if not tiny_shakespeare_path.exists():
        tiny_shakespeare_path.write_bytes(get(
            "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/
               tinyshakespeare/input.txt", timeout=5).content)
~~~

````
    return tiny_shakespeare_path


def ds_writingprompts(path="./data/") -> dict[str, tuple[Path, Path]]:
    """Returns a dataset object for interacting with the writing prompts dataset
        as extracted by Fan et al. (2015)

    Returns
    -------
    dict[str, tuple[Path, Path]]
        A dictionary object with the following structure:
        ‘‘‘
        - split: (source, target)
        ‘‘‘
        Where source is one of ‘[test, train, val]‘ and source and target are the
            "prompt" and "response(s)" files, respectively.

    Example Usage
    -------
    ‘‘‘
    ds = ds_writingprompts()
    with open(ds["train"][1]) as f:
        f.read()
    ‘‘‘
    """
    wppath = Path(path) / "writingPrompts/"
    if not wppath.exists():

        file = tarfile.open(fileobj=BytesIO(get(
            "https://dl.fbaipublicfiles.com/fairseq/data/writingPrompts.tar.gz",
               timeout=5).content))

        file.extractall(wppath.parent.parent)

    return {"test": (wppath / "test.wp_source", wppath / "test.wp_target"),
            "train": (wppath / "train.wp_source", wppath / "train.wp_target"),
            "val": (wppath / "valid.wp_source", wppath / "valid.wp_target")}


def ds_dgt(path="./data/") -> Path:
    """Returns the DGT-Acquis dataset offered by the European Union, etc.

    Parameters
    ----------
    path : str, optional
        Path to the data directory, by default "./data/"

    Returns
    -------
    Path
        A path reference for the available file that can be loaded next.
    """
    ds_path = Path(path) / "dgt" / "data.en.txt"
    if not ds_path.exists():
        with zipfile.ZipFile(BytesIO(get(
                "https://wt-public.emm4u.eu/Resources/DGT-Acquis-2012/data.en.txt
                    .zip", timeout=5).content)) as file:
            file.extractall(ds_path.parent)

    return ds_path
````

```
def load_imageability() -> Path:
    """
    Loads the imageability dataset from Cortese et al. (2004) and returns the
        path to the file.
    """

    im_path = Path(__file__).parent / "static" / \
        "imageability" / "cortese20004norms.csv"

    if not im_path.exists():
        with zipfile.ZipFile(BytesIO(get(r'https://static-content.springer.com/
            esm/art%3A10.3758%2FBF03195585/MediaObjects/Cortese-BRM-2004.zip',
            timeout=5).content)) as file:
             file.extractall(im_path.parent)

        for file in im_path.parent.glob('**/*'):
            file.rename(im_path.parent / file.name)

        # (im_path.parent / "Cortese-BRMIC-2004").rmdir()
    return im_path

def load_concreteness() -> Path:
    conc_path = Path(__file__).parent / "static" / \
        "concreteness" / "concreteness.csv"

    if not conc_path.exists():
        conc_path.parent.mkdir(exist_ok=True, parents=True)
        conc_path.write_bytes(get(r'http://crr.ugent.be/papers/
            Concreteness_ratings_Brysbaert_et_al_BRM.txt', timeout=5).content)

    return conc_path

def load_freq() -> Path:
    """_summary_

    Returns
    -------
    Path
        _description_
    """
    freq_path = Path(__file__).parent / "static" / \
        "frequency" / "frequency.csv"

    if not freq_path.exists():
        freq_path.parent.mkdir(exist_ok=True, parents=True)
        freq_path.write_bytes(get(r'https://ucrel.lancs.ac.uk/bncfreq/lists/1
            _1_all_alpha.txt', timeout=5).content)

    return freq_path

def read_texts(path: Path | str, length: int = 1_000_000) -> list[str]:
    """Returns a list of strings sequentially read from the path specified as the
         option.

    Parameters
    ----------
    path : Path
        Path to read from. The document will be opened in text-mode.
    length : int, optional
        The desired length of all texts, by default 1_000_000
    Returns
    -------
    list[str]
        List of the read character sequences.
    """
    with open(path) as f:
        text = []
        line = f.read(length)
        while len(line) > 0:
            text.append(line)
            line = f.read(length)
    return text

def load_machinetext(_path: str = './data/') -> dict:
    # Adapted from https://github.com/openai/gpt-2-output-dataset/blob/master/
        download_dataset.py
    path = Path(_path) / 'machinetext'
    if not path.exists():
        path.mkdir()

    _ds = [
        'webtext',
        'small-117M',  'small-117M-k40',
        'medium-345M', 'medium-345M-k40',
        'large-762M',  'large-762M-k40',
        'xl-1542M',    'xl-1542M-k40',
    ]

    _splits = [
        'train', 'valid', 'test'
    ]

    filenames = [f"{ds}.{split}.jsonl" for split in _splits for ds in _ds]

    # check if dir is empty
    if next(path.iterdir(), None) is None:

        for filename in filenames:
            r = get("https://openaipublic.azureedge.net/gpt-2/output-dataset/v1/"
                + filename, timeout=30, stream=True)

            with open(path / filename, 'wb') as f:
                file_size = int(r.headers["content-length"])
                chunk_size = 1000
                with tqdm(ncols=100, desc="Fetching " + filename, total=file_size
                    , unit_scale=True) as pbar:
                    # 1k for chunk_size, since Ethernet packet size is around
                        1500 bytes
                    for chunk in r.iter_content(chunk_size=chunk_size):
                        f.write(chunk)
                        pbar.update(chunk_size)

    return {
        ds: [path / f"{ds}.{split}.jsonl" for split in _splits] for ds in _ds
    }


if __name__ == "__main__":
    print("You should use the functions defined in the file, not run it directly!
        ")
```

Listing 8: Demo Notebook

```
# %%
import sys
import os.path
# Addition to path to unlock relative import to the madhatter package
sys.path.append(os.path.abspath(os.path.pardir))
import matplotlib.pyplot as plt

from madhatter.benchmark import CreativityBenchmark

# Read the text
with open('carroll-alice.txt') as f:
    text = f.read()

# Initialize the benchmark
bench = CreativityBenchmark(text, 'Alice in Wonderland')


# %%
# We have easy access methods for relevant segmentations of the text
bench.sents
bench.words
bench.tokenized_sents
bench.tagged_words
bench.tagged_sents


# %%
# We have easy access to things like frequency distributions over the whole book
bench.book_postag_counts()

# %%
# Similarly, we can make use of the metrics without having to create the
    benchmark object. If we choose to instead integrate with a NLP library like
    SpaCy.

from madhatter import metrics, utils
from nltk import word_tokenize

sent = "The quick brown fox jumped over the lazy dog."
metrics.concreteness(word_tokenize(sent), utils.get_concreteness_df()) # type:
    ignore

# %%
# Finally, we can put it all together by generating an overall Report object
    containing metrics for the whole text like so:

report = bench.report()

print(report)

# This object can later be used inside a machine learning pipeline to learn
    features about text to be used in classification and other tasks. See
    experiment.ipynb for examples.

# %%
# We also have access to a few different preset plotting functions

bench.plot_postag_distribution()
bench.plot_transition_matrix()

# %%
# We also have access to a variety of different metrics about the text:

conc = bench.concreteness_ratings()

# Shows all words along with their respective concreteness ratings
list(zip(bench.lemmas(), conc))

# %%
# We can also implement our own plots with the functions available to us

import matplotlib.pyplot as plt

plt.figure(figsize=(32,8))
plt.plot(conc)

# %%
# We can also showcase more advanced metrics utilizing LLMs:
# Note the spikes, those are moments in the context with high predictability. Predictability is a measure of the LLM's confidence in a given context.
# Low points signify low predictability -- that is, the expected word is not as predictable by the model, while high points mean that the model found less difficulty predicting the text.

plt.figure(figsize=(20,8))
plt.plot(report.predictability)

# %%
# Note the surprisal metric. It shows how similar or dissimilar potential contextual replacements are. It is strictly defined as the average of the top K likeliest replacements of the word in the given context. Higher scores mean that the word was expected and not too unusual. Lower scores mean that the word was "surprising" in this context, and suggested replacements had low or no similarity with the actual word being used.

plt.figure(figsize=(20,8))
plt.plot(report.surprisal)

# %%
%%timeit
list(i for i in range(10))

# %%
from madhatter.benchmark import BookReport

# an arbitary norm based on some observations for max possible values in data, can be improved
norm = BookReport(title='', nwords=20_000, mean_wl=6, mean_sl=300,
   mean_tokenspersent=40, prop_contentwords=0.10, mean_conc=5, mean_img=7,
   mean_freq=5, prop_pos=None, surprisal=None, predictability=None)

bench.plot_report(global_dist = norm,
    categories=["mean_wl", "mean_sl", "prop_contentwords", "mean_tokenspersent",
        "mean_conc", "mean_img", "mean_freq"],
    include_llm = False, print_time=False, include_pos=False
)

# %%
from madhatter.models import predict_tokens, default_model
```

```
model, tok = default_model()
res = predict_tokens("the quick brown fox jumped over the", "fox", model, tok,
    return_tokens=True)

# %%
import pandas as pd

print(pd.DataFrame(list(zip(res[0], res[1]))).to_latex(index=False)
)

# %%

import numpy as np

np.gradient(res[0])
```

Listing 9: Experiment Notebook

```
# %%
import sys
import os.path
# Addition to path to unlock relative import to the madhatter package
sys.path.append(os.path.abspath(os.path.pardir))

from madhatter import *
from madhatter.loaders import *

import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

from multiprocess.pool import Pool
# A progress bar to try to give an overall idea of the progress made.
from tqdm import tqdm
import pickle
from pathlib import Path

from sklearn.metrics import f1_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, precision_score, recall_score,
    accuracy_score
from sklearn.model_selection import GridSearchCV, PredefinedSplit,
    train_test_split
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler
from IPython.display import display

input_length = 100_000

print_latex = False


# %%
# nlp = spacy.load(”en_core_web_sm”, disable=[
#                    ”ner”,
#                   #  ”lemmatizer”,
#                    ”textcat”, ”attribute_ruler”])
# nlp.pipe_names
```

```
def read_texts(path: Path | str, length: int = 1_000_000) -> list[str]:
    """Returns a list of strings sequentially read from the path specified as the
        option.

    Parameters
    ----------
    path : Path
        Path to read from. The document will be opened in text-mode.
    length : int, optional
        The desired length of all texts, by default 1_000_000

    Returns
    -------
    list[str]
        List of the read character sequences.
    """
    with open(path) as f:
        text = []
        line = f.read(length)
        while len(line) > 0:
            text.append(line)
            line = f.read(length)
    return text


def split_strings(string: str, length=1_000_000):
    ret = []
    i = 0
    read = string[i*length:(i+1)*length]
    while len(read) > 0:
        ret.append(read)
        i += 1
        read = string[i*length:(i+1)*length]
    return ret


# %% [markdown]
# SpaCy performance concerns
#
# | Processes | Total Time (s) | Peak Total Memory (MB) |
# | --- | --- | --- |
# | 1 (SpaCy pipe) | 25.104 | 6487 |
# | 16 (SpaCy pipe) | 45.345 | 6340 |
# | 16 (multiprocessing) | 8.313 | 6679 |
#

# %% [markdown]
# ### Memory usage of Spacy vs Custom Package
# | Framework | peak memory | increment |
# |-------------|---------------|-------------|
# | Spacy | 5089.13 MiB |  4465.29 MiB |
# | Mad Hatter| 434.81 MiB  | 48.75 MiB |
#
# Increment here is the more important number as it tells us how memory usage
    peaks when performing a given operation.
#

# %% [markdown]
# ## Experimentation with pipelines
# Here we prepare a pipeline that will take the list of resources and return a
```

```
    list of ‘Report‘ objects. Those ‘Report‘ objects are then fed into a Pandas
    dataframe for further analysis. For better performance, we use the ‘
    multiprocessing‘ module to parallelize the pipeline, as each text is largely
    independent.
#
#
# Example listing of the pipeline:
# ‘‘‘python
# def pipeline(resources: list[str]):
#     reports = []
#     for resource in resources:
#         report = Report(resource)
#         reports.append(report)
#     return reports
# ‘‘‘

# %%
def process(file: str, title: str | None = None) -> BookReport:
    try:
        return CreativityBenchmark(file, title if title is not None else "unknown
            ").report(print_time=False, include_pos=True)
    except:
        return BookReport('')


def process_texts(args, processes: int = 16):
    ”””Note: args should be of the form (file, title if any)”””
    with Pool(processes) as p:
        return p.starmap(process, tqdm(args, total=len(args)))


def save_results(results, savepath):
    with open(savepath, 'wb') as file:
        pickle.dump(results, file)


def load_results(savepath):
    with open(savepath, 'rb') as file:
        return pickle.load(file)


# %% [markdown]
# ## Measuring the Gutenberg/Fiction dataset
# Note the lack of variety here. Gutenberg only has 18 works, but they lead to
    2124 texts of length 100000. This may be a somewhat flawed methodology so I
    recommend exploring more fictional works.

# %%
savepath_creative = Path("./results/creative.parquet")

if savepath_creative.exists():
    creative_df = pd.read_parquet(savepath_creative)
else:
    from nltk.corpus import gutenberg

    creative_fns = [file for file in gutenberg.fileids()]
    creative_files = []
    for file in creative_fns:
        listt = split_strings(gutenberg.raw(creative_fns), length=input_length)
        creative_files.extend([(_, file) for _ in listt])
    print(len(creative_files))

    creative_results = process_texts(creative_files)

    creative_df = pd.DataFrame(creative_results)

    creative_df.insert(creative_df.shape[1], "class", "PG")

    creative_df.to_parquet(savepath_creative)

creative_df.hist(bins=30)
creative_df.head()


# %% [markdown]
# ## Loading legal datasets into the pipeline

# %%
legal_path = Path("./results/legal.parquet")
if legal_path.exists():
    legal_df = pd.read_parquet(legal_path)
else:
    from nltk.corpus import europarl_raw

    legal_texts = read_texts(ds_dgt(), length=input_length)

    europarl_txt = split_strings("".join([" ".join(
        [" ".join(para) for para in chap]) for chap in europarl_raw.english.
           chapters()]), length=input_length)
    legal_texts.extend(europarl_txt)

    legal_results = process_texts(
        [(legal_text, f"legal_text_{i}") for i, legal_text in enumerate(
           legal_texts)])

    legal_df = pd.DataFrame(legal_results)
    legal_df.insert(legal_df.shape[1], "class", "LG")

    legal_df.to_parquet(legal_path)


legal_df.hist(bins=30)
legal_df.head()


# %% [markdown]
# ## Loading writing prompts

# %%
wp_savepath = Path("./results/wp.parquet")
if wp_savepath.exists():
    writingprompts_df = pd.read_parquet(wp_savepath)
else:
    # TODO: Possibly try out stuff like actually splitting the writingprompts
       dataset instead of reading continuous text.

    wp_path = ds_writingprompts()
    writingprompts = read_texts(wp_path["train"][1], length=input_length)
    writingprompts.extend(read_texts(wp_path["test"][1], length=input_length))
    writingprompts.extend(read_texts(wp_path["val"][1], length=input_length))
    # Length (100_000 chars) = 100089
    print(f"Length of writingprompts dataset: {len(writingprompts)}")


    wp_results = process_texts(
        list((_, f"writingprompts_{i}") for i, _ in enumerate(writingprompts)))


    # Whole thing took around 35 minutes on battery charge

    writingprompts_df = pd.DataFrame(wp_results)
    writingprompts_df.insert(writingprompts_df.shape[1], "class", "WP")

    writingprompts_df.to_parquet(wp_savepath)


writingprompts_df.hist(bins=30)
writingprompts_df.head()


# %% [markdown]
# A little visualization of what is happening behind the scenes. It seems like
    the novels have quite a bit more variety behind them at first glance.

# %% [markdown]
# ## Experiment
#

# %% [markdown]
# After running the pipeline, we concatenate the results into a single dataframe
    which we can then use for further analysis.

# %%
# Join
df = pd.concat([creative_df.head(2000), writingprompts_df.head(
    2000), legal_df.head(2000)], ignore_index=True)
df = df.join(pd.json_normalize(df["prop_pos"]).fillna(0.0))  # type: ignore

df = df.drop(columns=['predictability', 'surprisal'])


df["class"] = df["class"].astype('category')

def remove_outliers(df, deviation: float = 3) -> pd.DataFrame:
    # Remove outliers

    df = df.copy()

    cols = df.select_dtypes('number').columns
    df_sub = df.loc[:, cols]  # type: ignore
    lim = np.abs((df_sub - df_sub.mean()) / df_sub.std(ddof=0)) < deviation

    df.loc[:, cols] = df_sub.where(lim, np.nan)

    return df


df = remove_outliers(df, 3)

# fix bugged sentence length
df["mean_sl"] = df["mean_sl"].where(np.abs(
    (df["mean_sl"] - df["mean_sl"].mean()) / df["mean_sl"].std()) < 0.9, np.nan)

df = df.dropna()

# ####################

# drop unneeded columns and select features
xdf = df.drop(["title", "class", "prop_pos"], axis=1)

# drop arbitrary columns to see how results change
xdf = xdf.drop([
    # ”nwords”,
    # ”mean_wl”,
    # ”mean_sl”,

    # ”NOUN”,
    # ”ADJ”,
    # ”VERB”,
    # ”mean_conc”,
    # ”mean_img”,
    # ”mean_freq”,

    # ”prop_contentwords”,
    # ”mean_tokenspersent”
], axis=1)
ydf = df["class"]

# make the splits
xtrain, xtest, ytrain, ytest = train_test_split(xdf, ydf, train_size=0.8)
xtest, xval, ytest, yval = train_test_split(xtest, ytest, test_size=0.5)

# create the pipeline
model = Pipeline(steps=[("scaler", StandardScaler()),
                 ("logistic", LogisticRegression(max_iter=200))])
split = PredefinedSplit([-1]*len(xtrain)+[0]*len(xval))
params = {'logistic__C': [1/64, 1/32, 1/16,
                          1/8, 1/4, 1/2, 1, 2, 4, 8, 16, 32, 64]}
# search = GridSearchCV(model, params, cv=split,
#                       n_jobs=None, verbose=False, refit=False)
search.fit(pd.concat([xtrain, xval]), pd.concat([ytrain, yval]))
model = model.set_params(**search.best_params_)
model.fit(xtrain, ytrain)


ptrain = model.predict(xtrain)
pval = model.predict(xval)
ptest = model.predict(xtest)

experiment_dict = {
    'Experiment': 'Document Classification',
    'Size of Data': [
        len(xtrain),
        len(xval),
        len(xtest)
    ],
    'Accuracy': [
        accuracy_score(ytrain, ptrain),
        accuracy_score(yval, pval),
        accuracy_score(ytest, ptest)
    ],
    'Precision': [
        precision_score(ytrain, ptrain, average='macro'),
        precision_score(yval, pval, average='macro'),
        precision_score(ytest, ptest, average='macro')
    ],

    'Recall': [
        recall_score(ytrain, ptrain, average='macro'),
        recall_score(yval, pval, average='macro'),
        recall_score(ytest, ptest, average='macro')
    ],


    'F1-Score': [
        f1_score(ytrain, ptrain, average='macro'),
        f1_score(yval, pval, average='macro'),
        f1_score(ytest, ptest, average='macro')
    ],

}

experiment_df = pd.DataFrame(experiment_dict).T
experiment_df.columns = pd.MultiIndex.from_product(
    [['Split'], ['Train', 'Val', 'Test']])

experiment_df = experiment_df.drop(experiment_df.index[0])

display(experiment_df)

# if print_latex:
#     print(experiment_df.T.style
#             .format(precision=3)
#             .to_latex(hrules=True, position_float='centering',

#                       # type: ignore
#                       label=f'tab:{"_".join(experiment_dict["Experiment"].lower()
    .split())}',
#                       caption=f'Performance results for {experiment_dict["
    Experiment"]}',
#                       position='htbp'))


# hmap_path = f'./plots/document_classification/heatmap.png'
# cmap_aid = plt.subplots(dpi=300)
# sns.heatmap(confusion_matrix(model.predict(xtest), ytest), ax=cmap_aid[1])

# cmap_aid[0].savefig(hmap_path, bbox_inches='tight')


# %%
features = df.columns[df.columns.str.contains(
    "title|prop_pos|class") != True].to_list()

g = sns.pairplot(df, hue='class')
g.savefig('./plots/document_classification/big_distplot.png')


# %%
nrows = 3
ncols = 4

fig, axs = plt.subplots(nrows, ncols, figsize=(16, 11), dpi=200)
```

```
j = 0
for feature, ax in zip(features, axs.flatten()):
    g = sns.kdeplot(df, x=feature, ax=ax, hue='class', legend=False, fill=True)

# g.legend()
fig.legend(['WP', 'PG', 'LG'], loc='center right', fontsize='large')
axs[-1, -1].axis('off')

fig.savefig('./plots/distplots_classification/data_dist.png')


# %%
display_df = pd.DataFrame(
    model.coef_, columns=xdf.columns)  # type: ignore

display_df['categories'] = ydf.cat.categories

display(display_df.T)

# sns.catplot(display_df.T, x='' kind='bar')
# sns.barplot(display_df)
# plt.barh(display_df.index, display_df)


# %% [markdown]
# - Write about lemmatization approaches
# - Possibly make a diagram for how the process goes

# %% [markdown]
# ## Authorship Identification

# %%
number_authors = 1000
max_works = 30

# flag to turn on and off if works are to be split into chunks (default behaviour
    already takes a single chunk of `length` tokens)
chunks = False
pg_authorship_id_path = Path(
    f'./results/pgauthorship_{number_authors}.parquet')

if pg_authorship_id_path.exists():
    pg_df = pd.read_parquet(pg_authorship_id_path)
else:


    def open_pg(id: str):
        with open(f"./gutenberg/data/text/{id}_text.txt") as f:
            return f.read()

    csv = "./gutenberg/metadata/metadata.csv"
    pg = pd.read_csv(csv)

    authors = pg.groupby(['author'], group_keys=True).count(
    ).sort_values(by=['id'], ascending=False)['id']
    authors = authors.loc[authors.index.str.contains(
        r"Various|Anonymous|Unknown") != True]
    print(f"Uniquely identified authors in Project Gutenberg: {len(authors)}\n" +
          f"Uniquely identified pieces of literature: {len(pg)}")

    texts = {}
    for author in authors.index[:number_authors]:
        texts[author] = []
        for i, book in enumerate(list(pg.loc[pg["author"] == author].itertuples()
           )):
            if i > max_works:
                break
            texts[author].append(book.id)

    # for book in list(pg.itertuples())[:5]:
    #     print(book)
    filesnf = 0
    processing_set = []
    for author, collection in texts.items():
        for text in collection:
            try:
                if not chunks:
                    processing_set.append(
                        (open_pg(text)[:100_000], f"{text}_{author}"))
                else:
                    for i, t in enumerate(split_strings(open_pg(text), length=
                       input_length)):
                        processing_set.append((t, f"{text}_{i}_{author}"))
            except FileNotFoundError:
                filesnf += 1

    print(f"Files not found: {filesnf}")
    print(f"Total files: {sum(len(i) for i in texts.values())}")

    results = process_texts(processing_set)

    pg_df = pd.DataFrame(results)
    pg_df.insert(pg_df.shape[-1], "class", [_[-1]
                 for _ in pg_df["title"].str.split('_')])
    pg_df = pg_df.join(pd.json_normalize(
        pg_df["prop_pos"]).fillna(0.0))  # type: ignore

    pg_df.to_parquet(pg_authorship_id_path)

pg_df


# %%
# Distribution:
csv = "./gutenberg/metadata/metadata.csv"
pg = pd.read_csv(csv)

authors = pg.groupby(['author'], group_keys=True).count(
).sort_values(by=['id'], ascending=False)['id']
authors = authors.loc[authors.index.str.contains(
    r"Various|Anonymous|Unknown") != True]
print(f"Uniquely identified authors in Project Gutenberg: {len(authors)}\n" +
        f"Uniquely identified pieces of literature: {len(pg)}")

plt.plot(authors[:1000])
plt.ylabel('# works')
plt.xlabel('author')
plt.xticks(range(0,1001,100),range(0,1001,100));


# %% [markdown]
# ### Pipeline
# %%
# Join
df = pd.read_parquet(pg_authorship_id_path)

# drop unneeded columns and select features
xdf = df.drop(["title", "class", "prop_pos"], axis=1)

# drop arbitrary columns to see how results change
xdf = xdf.drop([
    # ”nwords”,
    # ”mean_wl”,
    # ”mean_sl”,

    # ”NOUN”,
    # ”ADJ”,
    # ”VERB”,
    # ”mean_conc”, ”mean_img”, ”mean_freq”,

    # ”prop_contentwords”,
    # ”mean_tokenspersent”
], axis=1)
ydf = df["class"]

# make the splits
xtrain, xtest, ytrain, ytest = train_test_split(xdf, ydf, train_size=0.8)
xtest, xval, ytest, yval = train_test_split(xtest, ytest, test_size=0.5)

# create the pipeline
model = Pipeline(steps=[("scaler", StandardScaler()),
                 ("logistic", LogisticRegression(max_iter=200))])
split = PredefinedSplit([-1]*len(xtrain)+[0]*len(xval))
params = {'logistic__C': [1/64, 1/32, 1/16,
                          1/8, 1/4, 1/2, 1, 2, 4, 8, 16, 32, 64]}
search = GridSearchCV(model, params, cv=split,
                      n_jobs=None, verbose=False, refit=False)
search.fit(pd.concat([xtrain, xval]), pd.concat([ytrain, yval]))
model = model.set_params(**search.best_params_)
model.fit(xtrain, ytrain)  # apply scaling on training data


ptrain = model.predict(xtrain)
pval = model.predict(xval)
ptest = model.predict(xtest)


experiment_dict = {
    'Experiment': f'Authorship Identification ({number_authors})',
    'Size of Data': [
        len(xtrain),
        len(xval),
        len(xtest)
    ],
    'Accuracy': [
        accuracy_score(ytrain, ptrain),
        accuracy_score(yval, pval),
        accuracy_score(ytest, ptest)
    ],
    'Precision': [
        precision_score(ytrain, ptrain, average='macro'),
        precision_score(yval, pval, average='macro'),
        precision_score(ytest, ptest, average='macro')
    ],

    'Recall': [
        recall_score(ytrain, ptrain, average='macro'),
        recall_score(yval, pval, average='macro'),
        recall_score(ytest, ptest, average='macro')
    ],


    'F1-Score': [
        f1_score(ytrain, ptrain, average='macro'),
        f1_score(yval, pval, average='macro'),
        f1_score(ytest, ptest, average='macro')
    ],

}

experiment_df = pd.DataFrame(experiment_dict).T
experiment_df.columns = pd.MultiIndex.from_product(
    [['Split'], ['Train', 'Val', 'Test']])

experiment_df = experiment_df.drop(experiment_df.index[0])

display(experiment_df)

if print_latex:
    print(experiment_df.T.style
          .format(precision=3)
          .to_latex(hrules=True, position_float='centering',

                    # type: ignore
                    label=f'tab:{"_".join(experiment_dict["Experiment"].lower().
                       split())}',
                    caption=f'Performance results for {experiment_dict["
                       Experiment"]}',
                    position='htbp'))

hmap_path = f'./plots/authorship_identification/aid_{number_authors}.png'
cmap_aid = plt.subplots(dpi=300)
sns.heatmap(confusion_matrix(model.predict(xtest), ytest), ax=cmap_aid[1])
cmap_aid[0].savefig(hmap_path, bbox_inches='tight')


# %%
# Join

from typing import Literal


dataset_type: Literal['-k40', ""] = "-k40"
model: Literal['xl-1542M', 'small-117M', 'large-762M','medium-345M'] = "xl-1542M"
nsamples = 40_000
mgtresultspath = Path(f'./results/mgt_results_{nsamples}_{model}{dataset_type}.
   parquet')


if mgtresultspath.exists():
    df_mgtresults = pd.read_parquet(mgtresultspath)
else:
    mgt_paths = load_machinetext()
    mgt = mgt_paths[model + dataset_type][0]
    non_mgt = mgt_paths["webtext"][0]

    with open(mgt) as f:
        mgt = pd.read_json(f, lines=True)

    with open(non_mgt) as f:
        non_mgt = pd.read_json(f, lines=True)

    mgt["class"] = "MGT"
    non_mgt["class"] = "HUMAN"
    df = pd.concat([mgt, non_mgt])
    df = df.reset_index()
    df["class"] = df["class"].astype('category')

    sample = pd.concat([df.loc[df["class"] =='MGT'].sample(nsamples//2), df.loc[
       df["class"] == 'HUMAN'].sample(nsamples//2)])

    results = process_texts(
        [(sample["text"][i], sample["class"][i]) for i in sample.index])

    df_mgtresults = pd.DataFrame(results)
    df_mgtresults["class"] = df_mgtresults["title"].astype('category')

    df_mgtresults.to_parquet(mgtresultspath)


# %%
# df = df.reset_index()
df = df_mgtresults

# drop unneeded columns and select features
xdf = df.drop(["title", "class", "prop_pos", "surprisal","predictability"], axis
   =1)

# drop arbitrary columns to see how results change
xdf = xdf.drop([
    # "nwords",
    # "mean_wl",
    # "mean_sl",

    # "NOUN",
    # "ADJ",
    # "VERB",
    # "mean_conc", "mean_img", "mean_freq",

    # "prop_contentwords",
    # "mean_tokenspersent"
], axis=1)
ydf = df["class"]

# make the splits
xtrain, xtest, ytrain, ytest = train_test_split(xdf, ydf, train_size=0.8,
   random_state=42)
xtest, xval, ytest, yval = train_test_split(xtest, ytest, test_size=0.5,
   random_state=42)

# create the pipeline
model = Pipeline(steps=[("scaler", StandardScaler()),
                 ("logistic", LogisticRegression(max_iter=200))])
```

```
split = PredefinedSplit([-1]*len(xtrain)+[0]*len(xval))
params = {'logistic__C': [1/64, 1/32, 1/16,
                          1/8, 1/4, 1/2, 1, 2, 4, 8, 16, 32, 64]}
search = GridSearchCV(model, params, cv=split,
                      n_jobs=None, verbose=False, refit=False)
search.fit(pd.concat([xtrain, xval]), pd.concat([ytrain, yval]))
model = model.set_params(**search.best_params_)
model.fit(xtrain, ytrain)  # apply scaling on training data

ptrain = model.predict(xtrain)
pval = model.predict(xval)
ptest = model.predict(xtest)

experiment_dict = {
    'Experiment': 'MGT Detection',
    'Size of Data': [
        len(xtrain),
        len(xval),
        len(xtest)
    ],
    'Accuracy': [
        accuracy_score(ytrain, ptrain),
        accuracy_score(yval, pval),
        accuracy_score(ytest, ptest)
    ],
    'Precision': [
        precision_score(ytrain, ptrain, average='macro'),
        precision_score(yval, pval, average='macro'),
        precision_score(ytest, ptest, average='macro')
    ],

    'Recall': [
        recall_score(ytrain, ptrain, average='macro'),
        recall_score(yval, pval, average='macro'),
        recall_score(ytest, ptest, average='macro')
    ],


    'F1-Score': [
        f1_score(ytrain, ptrain, average='macro'),
        f1_score(yval, pval, average='macro'),
        f1_score(ytest, ptest, average='macro')
    ],

}

experiment_df = pd.DataFrame(experiment_dict).T
experiment_df.columns = pd.MultiIndex.from_product(
    [['Split'], ['Train', 'Val', 'Test']])

experiment_df = experiment_df.drop(experiment_df.index[0])

display(experiment_df)

if print_latex:
    print(experiment_df.T.style
        .format(precision=3)
        .to_latex(hrules=True, position_float='centering',

                    # type: ignore
                    label=f'tab:{"_".join(experiment_dict["Experiment"].lower().
                       split())}',
                    caption=f'Performance results for {experiment_dict["
                       Experiment"]}',
                    position='htbp'))

hmap_path = f'./plots/mgt_detection/cmatrix_xl.png'
cmap_aid = plt.subplots(dpi=300)
sns.heatmap(confusion_matrix(model.predict(xtest), ytest,
            labels=ydf.cat.categories.tolist()), ax=cmap_aid[1], annot=True, fmt=
               "g", cbar=False)
cmap_aid[0].savefig(hmap_path, bbox_inches='tight')
```