

## Supplemental Material S2. Scripts used for data analyses.

### **data\_pipeline.py** – This script parses CHAT transcripts in xml format to a feature

```
#import xml.etree.ElementTree as ET
from copy import deepcopy
from dataclasses import replace
from logging import warning
from re import sub
import sys
from lxml import etree as ET
import csv
import json
import itertools
import data_checks
import numpy as np
import pandas as pd

import nltk
from nltk import word_tokenize
import gensim.downloader as api
from sklearn.linear_model import ElasticNetCV
from wordfreq import word_frequency

import pickle

from g2p_en import G2p

# PART 1

def check_subset(super, sub, pName=None):
    errors = set(super).difference(set(sub))
    if len(errors):
        if pName:
            print(f"Unexpected text found in {pName} tag")
        else:
            print("Unexpected text found")
    print(errors)

def extend_data(df, new_df):
    df = df.copy()
    new_df = new_df.copy()
    if len(df) == 0:
        new_start = 0
    else:
        new_start = int(df["location"].max()) + 1
    if len(new_df) > 0:
        new_df["location"] += new_start
    df = pd.concat([df, new_df], ignore_index=True)
    return df

def nstag(tag: str):
    ns = "http://www.talkbank.org/ns/talkbank"
```

```
    return f"{{{ns}}}{tag}"

def fresh_frame():
    return pd.DataFrame({
        "transcript_id":pd.Series(dtype='string'),
        "utterance_id": pd.Series(dtype='int'),
        "utterance":    pd.Series(dtype='int'),
        "location":     pd.Series(dtype='int'),
        "word":         pd.Series(dtype='string'),
        "pos":          pd.Series(dtype='string'),
        "errors":       pd.Series(dtype='object'),
    })

def parse_CHAT_tag(root: ET._Element, who = None, category = None,
replacement_mode = "swap", utt_id_start = None):
    df = fresh_frame()
    if utt_id_start is None:
        utt_id_start = 0
    if category:
        all_labels = list(root.findall(nstag("lazy-gem")))
        start_labels = [l for l in all_labels if l.get("label") ==
category]
        if len(start_labels):
            start_label = start_labels[0]
            start_index = root.index(start_label)
            if len(all_labels) > all_labels.index(start_label) + 1:
                end_index =
root.index(all_labels[all_labels.index(root[start_index]) + 1])
            else:
                end_index = len(root)
            utts = root[start_index:end_index]
        else:
            #raise ValueError(f"Category {category} not present in
tree.")
            warning(f"Category {category} not present in tree.\n")
            return df, utt_id_start
    else:
        utts = root[0:len(root)]

    utts = [u for u in utts if u.tag == nstag("u")]
    if who:
        utts = [u for u in utts if u.get("who") == who]
    datas = [df]
    for i, u in enumerate(utts):
        new_df = parse_u_tag(utter = u, replacement_mode =
replacement_mode)
        new_df["utterance_id"] = utt_id_start + i
        new_df["utterance"] = i
        datas.append(new_df)
    df = pd.concat(datas, ignore_index=True)
    return df, utt_id_start + len(utts)

def parse_u_tag(utter: ET._Element, replacement_mode = "swap"):
    df = fresh_frame()
```

```
# Check for anything unexpected
check_subset(utter.keys(), ["who", "uID", "u_attrb"])
check_subset([w.tag for w in utter],
              [nstag(tag) for tag in ["linker", "w", "e", "g", "pause",
"tagMarker", "postcode", "a", "s", "t", "media"]], "u_tag")

elms = list(utter)
for elm in elms:
    if elm.tag == nstag("w"):
        new_df = parse_w_tag(elm, replacement_mode)
        df = extend_data(df, new_df)
    elif elm.tag == nstag("g"):
        new_df = parse_g_tag(elm)
        df = extend_data(df, new_df)
    elif elm.tag == nstag("linker"):
        continue
    elif elm.tag == nstag("postcode"):
        continue
    elif elm.tag == nstag("tagmarker"):
        continue
    elif elm.tag == nstag("e"):
        continue
    elif elm.tag == nstag("pause"):
        continue
    elif elm.tag == nstag("s"):
        continue
    elif elm.tag == nstag("t"):
        continue
    elif elm.tag == nstag("media"):
        continue

return df

def parse_w_tag(word: ET._Element, replacement_mode = "swap"):
    df = fresh_frame()

    replacement_modes = ["ignore", "swap", "record"]
    if replacement_mode not in replacement_modes:
        raise ValueError(f"split_mode {replacement_mode} not in
{replacement_modes}")

    # Check for anything unexpected
    check_subset(word.keys(), ["type", "formType", "untranscribed",
"w_attrb"])
    check_subset([w.tag for w in word],
                  [nstag(tag) for tag in ["shortening", "p", "replacement", "mor",
"wk", "langs", "segment-repetition"]], "w_tag")

    error_infos = []

    if word.findall(nstag("replacement")):
        replacement = word.findall(nstag("replacement"))[0]
        replace_word_tag = replacement.findall(nstag("w"))[0]
```

```
        if replacement_mode == "swap":
            spoken_words_info_df = parse_w_tag(word,
replacement_mode="ignore")
            spoken_words = spoken_words_info_df
            intended_word = replace_word_tag
            word = intended_word
            error_infos.append({"error": "replacement", "original":
spoken_words})
        elif replacement_mode == "record":
            replacement_word_info_df = parse_w_tag(replace_word_tag,
replacement_mode="ignore")
            replacement_words = replacement_word_info_df
            error_infos.append({"error": "replacement", "replacements":
replacement_words})
        elif replacement_mode == "ignore":
            pass

    type = word.get("type")
    formtype = word.get("formType")
    untranscribed = word.get("untranscribed")
    text = ""
    alt_pos = "None"

    if word.text:
        text += word.text

    if word.findall(nstag("shortening")):
        for short in word.findall(nstag("shortening")):
            shortening = short.text
            text += short.text
            if short.tail:
                text += short.tail

    if word.findall(nstag("wk")): # Some words need hyphens, some need a
space, and some no space
        for part in word.findall(nstag("wk")):
            if part.text:
                text += part.text
            if part.tail:
                text += part.tail

    if word.findall(nstag("p")):
        for pros in word.findall(nstag("p")):
            prosidy = pros.text
            if pros.text:
                text += pros.text
            if pros.tail:
                text += pros.tail
            if pros.get("type") == "drawl":
                pass
            if pros.get("type") == "pause":
                pass

    if word.findall(nstag("langs")):
```

```
pass

if word.findall(nstag("segment-repetition")):
    for rep in word.findall(nstag("segment-repetition")):
        rep_text = rep.get("text")

if formtype:
    if formtype == "babbling":
        pass
    if formtype == "UNIBET":
        error_infos.append({"error": "UNIBET"})
        pass
    if formtype == "neologism":
        error_infos.append({"error": "neologism"})
        pass
    if formtype == "letter":
        pass
    if formtype == "quoted metareference":
        pass
    if formtype == "onomatopoeia":
        pass

if type:
    if type == "incomplete":
        alt_pos = "incomplete"
        pass
    if type == "fragment":
        alt_pos = "fragment"
        pass
    if type == "filler":
        alt_pos = "filler"
        pass

if untranscribed:
    error_infos.append({"error": "untranscribed", "reason":
untranscribed})
    if untranscribed == "unintelligible":
        pass
    if untranscribed == "untranscribed":
        pass

if not text:
    text = "???"
text = text.lower()
df = pd.concat([df, pd.DataFrame({"word": text, "errors":
[error_infos]})], ignore_index=True)

# Get part of speech

if word.findall(nstag("mor")):
    mor = word.findall(nstag("mor"))[0]
    pos_s = parse_mor_tag(mor)

    if len(pos_s) > 1:
```

```
df = fresh_frame()
texts = [" " + t for t in text.split(" ")]
texts[0] = texts[0].strip(" ")
if len(texts) == len(pos_s):
    words = [{"word": t} for t in texts]
else:
    words = [{"word": text + "_" + str(pos)} for pos in
pos_s]

words[0]["errors"] = error_infos
for w in words:
    df = pd.concat([df, pd.DataFrame(w)], ignore_index=True)
else:
    pos_s = [alt_pos]

for i, pos in enumerate(pos_s):
    df.iloc[i, df.columns.get_loc("pos")] = pos
    df.iloc[i, df.columns.get_loc("location")] = i

return df

def parse_g_tag(group: ET._Element):
    # Check for surprises
    check_subset(group.keys(), [], "g_attrb")
    check_subset([w.tag for w in group],
    [nstag(tag) for tag in ["g", "w", "error", "k", "ga", "tagMarker",
    "pause", "e", "overlap", "r"]], "g_tag")

    df = fresh_frame()

    for elm in group:
        type = elm.tag
        if type == nstag("g"):
            new_df = parse_g_tag(elm)
            df = extend_data(df, new_df)
        elif type == nstag("w"):
            new_df = parse_w_tag(elm)
            df = extend_data(df, new_df)
        elif type == nstag("error"):
            df.iloc[-1, df.columns.get_loc("errors")].append({"error":
"tag", "code": elm.text})
        elif type == nstag("k"):
            if elm.get("type") == "retracing":
                pass
            if elm.get("type") == "retracing with correction":
                pass
        elif type == nstag("ga"):
            if elm.get("type") == "muttering":
                pass
            if elm.get("type") == "whispering":
                pass
            if elm.get("type") == "singsong":
                pass
        elif type == nstag("tagMarker"):
            pass
```

```
        elif type == nstag("pause"):
            pass
        elif type == nstag("e"):
            pass
        elif type == nstag("overlap"):
            pass # Can be present instead of k or error
        elif type == nstag("r"):
            count = int(elm.get("times"))
            rep_data = df.copy()
            for i in range(0, count - 1):
                df = extend_data(df, rep_data)
    return df

def parse_mor_tag(mor: ET._Element, split_mode:str="single",
sub_mode:str="ignore"):
    sub_modes = ["concat", "ignore"]
    split_modes = ["single", "multiple"]
    if split_mode not in split_modes:
        raise ValueError(f"split_mode {split_mode} not in {split_modes}")
    if sub_mode not in sub_modes:
        raise ValueError(f"sub_mode {sub_mode} not in {sub_modes}")
    # May be mwc rather than mw
    mor_list = [mor]
    pos_s = []
    while mor_list:
        mor = mor_list[0]
        if mor.findall(nstag("mor-post")) and (split_mode == "multiple"):
            new_mor = list(mor.findall(nstag("mor-post")))
            new_mor.extend(mor_list)
            mor_list = new_mor
        mor_list.remove(mor)
        check_subset(mor.keys(), ["type"], "mor_attb")
        check_subset([w.tag for w in mor],
            [nstag(tag) for tag in ["mw", "gra", "mwc", "mor-post", "mt",
"menx"]], "mor_tag")
        # May be mwc rather than mw
        if mor.findall(nstag("mw")):
            mw = mor.findall(nstag("mw"))[0]
            check_subset(mw.keys(), [], "mw_attb")
            check_subset([w.tag for w in mw],
                [nstag(tag) for tag in ["pos", "stem", "mk", "mpfx"]],
"mw_tag")
            elif mor.findall(nstag("mwc")):
                mw = mor.findall(nstag("mwc"))[0]
                check_subset(mw.keys(), [], "mwc_attb")
                check_subset([w.tag for w in mw],
                    [nstag(tag) for tag in ["pos", "stem", "mk", "mpfx", "mw"]],
"mwc_tag")
            else: print("mor missing mw or mwc")
            pos = mw.findall(nstag("pos"))[0]
            check_subset(pos.keys(), [], "pos_attb")
            check_subset([w.tag for w in pos],
                [nstag(tag) for tag in ["s", "c"]], "pos_tag")
            c = pos.findall(nstag("c"))[0]
```

```
    check_subset(c.keys(), [], "c_attrb")
    check_subset([w.tag for w in c],
        [nstag(tag) for tag in []], "c_tag")
    pos_str = c.text
    if sub_mode == "concat" and pos.findall(nstag("s")):
        s = pos.findall(nstag("s"))[0]
        check_subset(s.keys(), [], "s_attrb")
        check_subset([w.tag for w in s],
            [nstag(tag) for tag in []], "s_tag")
        pos_str += " " + s.text
    pos_s.append(pos_str)
    return pos_s

def parse_people_info(csv_filename, attbs):
    with open(csv_filename) as file:
        people_full = list(csv.DictReader(file))
        people_extra_df = pd.read_excel('Data/english-results-data.xlsx',
            sheet_name="Time 1")
        people_extra_dict = dict(zip(people_extra_df["Participant ID"],
            people_extra_df["WAB AQ"]))
        people_extra_dict = {part_id.upper(): attbs for part_id, attbs in
            people_extra_dict.items() if pd.notnull(part_id)}
        people_extra_dict_umd = {part_id[4:]: attbs for part_id, attbs in
            people_extra_dict.items() if "MMA" in part_id or "MMB" in part_id}
        people_extra_dict = people_extra_dict | people_extra_dict_umd
        people = []
        # disc_map = {}
        # for a, properties in attbs.items():
        #     a_name = properties["name"]
        #     a_type = properties["type"]
        #     if a_type != "discrete":
        #         continue
        #     a_map = {}
        #     disc_map[a_name] = a_map
        #     for p in people_full:
        #         if p[a] not in a_map:
        #             a_map[p[a]] = len(a_map)
        for p in people_full:
            new_p = {"transcript_id": p["Participant ID"]}
            new_p["filename"] = f"Data/Transcripts-
xml/{new_p['transcript_id']}.xml"
            for a, properties in attbs.items():
                a_name = properties["name"]
                a_type = properties["type"]
                if a_type == "continuous":
                    try:
                        new_p[a_name] = float(p[a])
                    except ValueError as e:
                        if p[a] == "U":
                            pass
                        elif p[a] == "":
                            pass
                        else:
                            raise e
```

```
        elif a_type == "discrete":
            if p[a] == "U":
                pass
            elif p[a] == "":
                pass
            else:
                new_p[a_name] = p[a]
    try:
        new_p["severity"] =
float(people_extra_dict[new_p["transcript_id"].upper()])
        new_p["severity_very_severe"] = new_p["severity"] <= 25.0
        new_p["severity_severe"] = 25.0 < new_p["severity"] and
new_p["severity"] <= 50.0
        new_p["severity_moderate"] = 50.0 < new_p["severity"] and
new_p["severity"] <= 75.0
        new_p["severity_mild"] = 75.0 < new_p["severity"] and
int(new_p["severity"]) <= 93.0
    except ValueError as e:
        new_p["severity"] = None
        new_p["severity_very_severe"] = False
        new_p["severity_severe"] = False
        new_p["severity_moderate"] = False
        new_p["severity_mild"] = False
    people.append(new_p)
    df = pd.DataFrame(people, columns=["transcript_id", "filename"] +
[attb["name"] for attb in list(attbs.values())]
        + ["severity", "severity_very_severe",
"severity_severe", "severity_moderate", "severity_mild"])
    return df

def add_db_features(df):
    db_atts = {}
    with open("Data/bird database.tsv", "r") as db:
        word_list = list(csv.DictReader(db, delimiter="\t"))
        db_atts = {w["Word"]:w for w in word_list}
    def retrieve(word, attribute):
        val = db_atts.get(word, {}).get(attribute, None)
        val = float(val) if val is not None and val != "." else None
        return val
    df["imageability 1"] = df.apply(lambda row: retrieve(row["word"],
"new_Imageability"), axis=1)
    df["age of acquisition 1"] = df.apply(lambda row:
retrieve(row["word"], "new_AoA"), axis=1)
    df["frequency 1"] = df.apply(lambda row: retrieve(row["word"],
"Frequency"), axis=1)

def add_db2_features(df):
    db_atts = {}
    with open("Data/mrc2.dct", "r") as db:
        for line in db:
            a = {}
            s = line.split("|")
            clean = lambda a: int(a) if int(a) != 0 else None
            a["WORD"] = s[0][51:].lower()
```

```
a["NLET"] = clean(s[0][0:2])
a["NPHON"] = clean(s[0][2:3])
a["NSYL"] = clean(s[0][4])
a["K-F-FREQ"] = clean(s[0][5:10])
a["T-L-FREQ"] = clean(s[0][15:21])
a["BROWN-FREQ"] = clean(s[0][21:25])
a["FAM"] = clean(s[0][25:28])
a["CONC"] = clean(s[0][28:31])
a["IMAG"] = clean(s[0][31:34])
a["AOA"] = clean(s[0][40:43])

db_atts[a["WORD"]] = a

def retrieve(word, attribute):
    val = db_atts.get(word, {}).get(attribute, None)
    val = float(val) if val is not None and val != "." else None
    return val
df["imageability 2"] = df.apply(lambda row: retrieve(row["word"],
"IMAG"), axis=1)
df["age of acquisition 2"] = df.apply(lambda row:
retrieve(row["word"], "AOA"), axis=1)
df["frequency 2"] = df.apply(lambda row: retrieve(row["word"],
"BROWN-FREQ"), axis=1)

def add_letters(df):
    df["letter"] = df.apply((lambda row: len(row["word"]) if
pd.notnull(row["word"]) else None), axis=1)

def transcript_str(df: pd.DataFrame, to_print=["word"],
color_errors=True, to_print_errors=[], to_print_person = [],
print_head=True, min_size=0):
    result_str = ""
    ERROR = '\033[93m'
    ERROR2 = '\033[91m'
    END_ERROR = '\033[0m'
    for utter_num, utter_df in df.groupby("utterance_id"):
        utter_df = utter_df.sort_values("location")
        lines = [""] * len(to_print)
        if to_print_errors:
            lines.append("")
        if print_head:
            size = min_size
            for type in to_print:
                size = max(size, len(str(type)))
            if to_print_errors:
                size = max(size, len("errors"))
            for i, type in enumerate(to_print):
                lines[i] += f"{str(type) + ':'<{size+1}} "
            if to_print_errors:
                lines[-1] += f{'errors':<{size+1}} "
        for word_index, word_row in utter_df.iterrows():
            has_error = False
            error_code = ""
            if color_errors:
```

```
        if len(word_row["errors"]) == 0:
            has_error = True
    if to_print_errors:
        for error in word_row["errors"]:
            if error_code:
                error_code += " "
            if "error" in to_print_errors:
                error_code += error["error"]
            for err_atb in to_print_errors:
                if err_atb == "error":
                    continue
                err_atb_list = err_atb.split(">")
                target = error
                for sub_err_atb in err_atb_list:
                    if isinstance(target, dict) and sub_err_atb
in target:
                    target = target[sub_err_atb]
                    continue
                try:
                    if isinstance(target, list) and
int(sub_err_atb) < len(target):
                        target = target[int(sub_err_atb)]
                        continue
                    except ValueError:
                        pass
                    break
                else: # If the target is found
                    error_code += "[" + err_atb + ": " +
str(target) + "]"
                size = min_size
                for i, type in enumerate(to_print):
                    size = max(size, len(str(word_row[type])))
                if to_print_errors:
                    size = max(size, len(error_code))
                for i, type in enumerate(to_print):
                    if has_error:
                        lines[i] += f"{ERROR2}{str(word_row[type]):<{size}}
{END_ERROR}"
                    else:
                        lines[i] += f"{str(word_row[type]):<{size}} "
                if to_print_errors:
                    if has_error:
                        lines[-1] += f"{ERROR2}{str(error_code):<{size}}
{END_ERROR}"
                    else:
                        lines[-1] += f"{str(error_code):<{size}} "
                person_lines = []
                for person_feature in to_print_person:
                    person_lines.append(f"{person_feature}:
{utter_df[person_feature][-1]}")
                lines = person_lines + lines
                for line in lines:
                    result_str += line + "\n"
                result_str += "\n"
```

```
    return result_str

def count_valid_str(df: pd.DataFrame, to_print=["word"],
print_errors=False, print_head=True, min_size=0): #TODO fix later
    result_str = ""
    headers = []
    body = []
    sizes = []
    if print_head:
        for attr in to_print:
            sizes.append(max(min_size, len(attr)))
            headers.append(str(attr))
        if print_errors:
            sizes.append(max(min_size, len("errors")))
            headers.append("errors")
    for i, attr in enumerate(to_print):
        count = data_checks.count_attr(df, attr, target=[None],
invert=True, all=True)
        body.append(count)
        sizes[i] = max(sizes[i], len(str(count)))
    if print_errors:
        count = data_checks.count_err(df)
        body.append(count)
        sizes[-1] = max(sizes[-1], len(str(count)))
    if print_head:
        for s, h in zip(sizes, headers):
            result_str += f"{str(h):<{s}} "
        result_str += "\n"
    for s, b in zip(sizes, body):
        result_str += f"{str(b):<{s}} "
    return result_str

def construct_all_utterances():
    # Get list of people
    person_csv_name = "Data/person-data.csv"
    attbs_json_name = "Data/attbs.json"
    people_attbs = {}
    with open(attbs_json_name, "r") as attbs_file:
        people_attbs = json.load(attbs_file)
    people_df = parse_people_info(person_csv_name, people_attbs)

    print(people_df)

    # Parse transcripts into list of
    transcripts = []
    start Utt_id = 0
    for person_index, person_row in people_df.iterrows():
        transcript_id = person_row["transcript_id"]
        filename = person_row["filename"]
        print(filename)
        tree = None
        with open(filename, encoding='utf8') as xml_file:
            tree = ET.parse(xml_file)
            if tree == None:
```

```
        raise Exception(f"No tree parsed for {filename}!")
    root = tree.getroot()
    transcript_df, start_utt_id = parse_CHAT_tag(root, who="PAR",
category="Cinderella", replacement_mode = "swap",
utt_id_start=start_utt_id)
    transcript_df["transcript_id"] = transcript_id
    if len(transcript_df) == 0:
        print(f"Missing {filename} Cinderella story.")
    #transcript = parse_CHAT_tag(root)
    transcripts.append(transcript_df)
    df = pd.concat(transcripts, ignore_index=True)
    # Process data
    add_db_features(df)
    add_db2_features(df)
    df["age of acquisition c"] = df.apply((lambda row: row["age of
acquisition 2"] if pd.isnull(row["age of acquisition 1"]) else row["age
of acquisition 1"]), axis=1)
    add_letters(df)
    df = df.merge(people_df, how='left', left_on="transcript_id",
right_on="transcript_id")

    errors = set()
    df["word_id"]=df.index
    for word_id, word in df.iterrows():
        if len(word["errors"]):
            errors = errors.union([e["error"] for e in word["errors"]])
    for error in errors:
        check_for_error = lambda row: error in [e["error"] for e in
row["errors"]]
        df[f"error_{error}"] = df.apply(check_for_error, axis=1)
        check_s = lambda row: True in [(e["error"] == "tag" and (e["code"] is
not None and e["code"][0] == 's')) for e in row["errors"]]
        df["error_tag_s"] = df.apply(check_s, axis=1)
        check_p = lambda row: True in [(e["error"] == "tag" and (e["code"] is
not None and e["code"][0] == 'p')) for e in row["errors"]]
        check_p_not_s = lambda row: ( True in [(e["error"] == "tag" and
(e["code"] is not None and e["code"][0] == 'p')) for e in row["errors"]]
and not True in [(e["error"] == "tag"
and (e["code"] is not None and e["code"][0] == 's')) for e in
row["errors"]] )
        df["error_tag_p"] = df.apply(check_p_not_s, axis=1)
        check_n = lambda row: True in [(e["error"] == "tag" and (e["code"] is
not None and e["code"][0] == 'n')) for e in row["errors"]]
        df["error_tag_n"] = df.apply(check_n, axis=1)
        check_m = lambda row: True in [(e["error"] == "tag" and (e["code"] is
not None and e["code"][0] == 'm')) for e in row["errors"]]
        df["error_tag_m"] = df.apply(check_m, axis=1)
        df.to_excel("Data/1_all_utters.xlsx", "all")
        df.to_pickle("Data/1_all_utters.pk1")

def main_1():
    construct_all_utterances()

# PART 2
```

```
def incorporate_from_excel(df:pd.DataFrame, sheet_name:str,
excel_index:str, feature_excel_map:dict, check_col:str=None):
    new_vals = pd.read_excel('Data/all_utters_extended.xlsx',
sheet_name=sheet_name)
    if check_col is not None:
        new_vals = new_vals[new_vals[check_col] == "Y"]
        feature_excel_map = {excel_index:"word_id", **feature_excel_map}
        new_vals = new_vals[list(feature_excel_map.keys())]
        new_vals = new_vals.rename(feature_excel_map, axis=1)
        temp_merge_df = df.merge(new_vals, how='left', on="word_id",
suffixes=("_x", None))
        if(df.shape[0] != temp_merge_df.shape[0]):
            print(f"MAJOR WARNING, {temp_merge_df.shape[0] - df.shape[0]}
ROWS ARE NOT ASSOCIATED WITH A WORD!!!")
        df.update(temp_merge_df)
        return df

def rework_pos(df:pd.DataFrame):
    pos_remap = {
        "filler": "interjection",
        "co": "interjection",
        "n": "noun",
        "cop": "verb",
        "det": "determiner",
        "adj": "adjective",
        "adv": "adverb",
        "None": None,
        "fragment": "OMIT",
        "inf": "infinitive",
        "v": "verb",
        "qn": "determiner",
        "coord": "conjunction",
        "pro": "pronoun",
        "prep": "preposition",
        "bab": "onomatopoeia",
        "part": "verb",
        "aux": "aux verb",
        "post": "verb",
        "mod": "verb",
        "conj": "conjunction",
        "comp": "complementizer",
        "neg": "adverb",
        "neo": None,
        "on": "onomatopoeia",
        "incomplete": "OMIT",
        "uni": None,
        "meta": "noun",
        "L2": "interjection",
        "sing": "onomatopoeia",
    }
    # print size of df
    print(df.shape)
    for old, new in pos_remap.items():
```

```
        df.loc[df["pos"] == old, "pos"] = new
    print(df.shape)
    df.loc[df["pos"].isna(), "pos"] = None
    print(df.shape)
    df = df[(df["pos"] != "OMIT") | df["pos"].isna()]
    print(df.shape)
    return df

def main_2():
    df:pd.DataFrame = pd.read_pickle("Data/1_all_utters.pkl")
    incorporate_from_excel(df, "neologisms replacements final", "Index",
{"Manual replacement": "word", "manual POS": "pos"}, "Problem Free")
    incorporate_from_excel(df, "semantic replacements final", "Index",
{"Manual replacement": "word"}, "Problem Free")
    df = rework_pos(df)
    df.to_excel("Data/2_all_utters_plus.xlsx", "all")
    df.to_pickle("Data/2_all_utters_plus.pkl")

# PART 3

def get_bird_db():
    img = {}
    aoa = {}
    freq = {}
    with open("Data/bird database.tsv", "r") as db:
        word_list = list(csv.DictReader(db, delimiter="\t"))
        for w in word_list:
            word = w["Word"].lower()
            imgval = w["new_Imageability"]
            aoaval = w["new_AoA"]
            freqval = w["Frequency"]
            img[word] = float(imgval) if imgval is not None and imgval !=
"." else None
            aoa[word] = float(aoaval) if aoaval is not None and aoaval !=
"." else None
            freq[word] = float(freqval) if freqval is not None and
freqval != "." else None
        return {"imageability 1":img, "age of acquisition 1":aoa, "frequency
1":freq}

def get_mrc2_db():
    img = {}
    aoa = {}
    freq = {}
    with open("Data/mrc2.dct", "r") as db:
        for line in db:
            a = {}
            s = line.split("|")
            clean = lambda a: int(a) if int(a) != 0 else None
            a["WORD"] = s[0][51:].lower()
            a["NLET"] = clean(s[0][0:2])
            a["NPHON"] = clean(s[0][2:3])
            a["NSYL"] = clean(s[0][4])
            a["K-F-FREQ"] = clean(s[0][5:10])
```

```
a["T-L-FREQ"] = clean(s[0][15:21])
a["BROWN-FREQ"] = clean(s[0][21:25])
a["FAM"] = clean(s[0][25:28])
a["CONC"] = clean(s[0][28:31])
a["IMAG"] = clean(s[0][31:34])
a["AOA"] = clean(s[0][40:43])

word = a["WORD"]
img[word] = a["IMAG"]
aoa[word] = a["AOA"]
freq[word] = a["BROWN-FREQ"]

return {"imageability 2": img, "age of acquisition 2": aoa,
        "frequency 2": freq}

def load_word_vectors():
    wv = api.load('word2vec-google-news-300')
    # wv = api.load('conceptnet-numberbatch-17-06-300')
    # wv = api.load('glove-wiki-gigaword-50')
    return wv

def main_3(save=True):
    print("Loading models")
    with open("Models/imageability 2_lr_model.pkl", 'rb') as file:
        imag_model: ElasticNetCV = pickle.load(file)
    with open("Models/age of acquisition 2_lr_model.pkl", 'rb') as file:
        aoa_model: ElasticNetCV = pickle.load(file)
    print("Load transcript data")
    df:pd.DataFrame = pd.read_pickle("Data/2_all_utters_plus.pkl")
    print("Loading word vectors")
    wv = load_word_vectors()
    print("Getting wordvectors")
    words = {}
    for word in df["word"]:
        wvec = None
        if word in wv:
            wvec = wv[word]
            words[word] = wvec
        elif word.title() in wv:
            wvec = wv[word.title()]
            words[word] = wvec
        else:
            pass
    words_list = list(words.keys())
    words_vecs = list(words.values())
    print("Predicting imputing values")
    words_img = imag_model.predict(words_vecs)
    words_aoa = aoa_model.predict(words_vecs)
    words_img_dict = {word: img for word, img in zip(words_list,
words_img)}
    words_aoa_dict = {word: aoa for word, aoa in zip(words_list,
words_aoa)}

    img_init = np.sum(pd.notna(df["imageability 2"]))
    aoa_init = np.sum(pd.notna(df["age of acquisition 2"]))
```

```
df["imageability 3"] = df.apply(lambda x: x["imageability 2"] if
pd.notna(x["imageability 2"]) else words_img_dict.get(x["word"], None),
axis=1)
df["age of acquisition 3"] = df.apply(lambda x: x["age of acquisition
2"] if pd.notna(x["age of acquisition 2"]) else
words_aoa_dict.get(x["word"], None), axis=1)
df["frequency 3"] = df.apply(lambda x: word_frequency(x["word"],
lang='en'), axis=1)
img_final = np.sum(pd.notna(df["imageability 3"]))
aoa_final = np.sum(pd.notna(df["age of acquisition 3"]))
freq_final = np.sum(df["frequency 3"] >= 0.000000000001)
img_impute = img_final - img_init
aoa_impute = aoa_final - aoa_init
all_total = len(df["word"])
print(f"Started with {img_init} img, imputed {img_impute} to get
{img_final}, missing only {all_total - img_final} out of {all_total}
total.")
print(f"Started with {aoa_init} aoa, imputed {aoa_impute} to get
{aoa_final}, missing only {all_total - aoa_final} out of {all_total}
total.")
print(f"freq has {freq_final}, missing only {all_total - freq_final}
out of {all_total}.")
pos_final = np.sum(pd.notna(df["pos"]))
print(f"pos has {pos_final}, missing only {all_total - pos_final} out
of {all_total}.")
df["letter"] = df.apply((lambda row: len(row["word"]) if
pd.notnull(row["word"]) else None), axis=1)
print("Letters updated")
if save:
    print("Saving files")
    df.to_excel("Data/3_all_utters_impute.xlsx", "all")
    df.to_pickle("Data/3_all_utters_impute.pkl")
```

# PART 3\_5

```
pos_universal_precedence = {
    "NOUN": 1,
    "VERB": 2,
    "ADJ": 3,
    "ADV": 4,
    "PRON": 5,
    "DET": 6,
    "ADP": 7,
    "CONJ": 8,
    "PRT": 9,
    "NUM": 10,
    ".": 11,
    "X": 12,
    None: 13
}
```

```
pos_universal_translation = {
    "NOUN": "NOUN",
    "VERB": "VERB",
```

```
"ADJ": "ADJ",
"ADV": "ADV",
"PRON": "PRON",
"DET": "DET",
"ADP": "ADP",
"CONJ": "CONJ",
"PRT": "PRT",
"NUM": "NUM",
".": ".",
"X": "X",
None: None
}

pos_precedence = {
    "noun": 1,
    "pronoun": 2,
    "verb": 3,
    "adjective": 4,
    "adverb": 5,
    "determiner": 6,
    "preposition": 7,
    "conjunction": 8,
    "infinitive": 9,
    "interjection": 10,
    "aux verb": 11,
    "complementizer": 12,
    "onomatopoeia": 13,
    None: 14,
}

pos_translation = {
    "CC": "conjunction",
    "CD": "determiner",
    "DT": "determiner",
    "EX": "pronoun", # adverb
    "FW": "interjection",
    "IN": "preposition",
    "JJ": "adjective",
    "JJR": "adjective",
    "JJS": "adjective",
    "LS": None,
    "MD": "verb",
    "NN": "noun",
    "NNS": "noun",
    "NNP": "noun", # interjection
    "NNPS": "noun", # interjection
    "PDT": "determiner",
    "POS": None,
    "PRP": "pronoun",
    "PRP$": "pronoun", # determiner
    "RB": "adverb",
    "RBR": "adverb",
    "RBS": "adverb", # determiner
    "RP": "adverb",
```

```
"SYM": None,
"TO": "infinitive",
"UH": "interjection",
"VB": "verb",
"VBD": "verb",
"VBG": "verb",
"VBN": "verb",
"VBP": "verb",
"VBZ": "verb",
"WDT": "pronoun",
"WP": "pronoun",
"WP$": "pronoun",
"WRB": "pronoun", # conjunction
".": None,
}

def pos_universal_resolution(tagged_tokens:list):
    pos = None
    for tagged_token in tagged_tokens:
        check = pos_universal_translation[tagged_token[1]]
        if pos is None:
            pos = check
        elif pos_universal_precedence[check] <
pos_universal_precedence[pos]:
            pos = check
    return pos

def pos_resolution(tagged_tokens:list):
    pos = None
    for tagged_token in tagged_tokens:
        check = pos_translation[tagged_token[1]]
        if pos is None:
            pos = check
        elif pos_precedence[check] < pos_precedence[pos]:
            pos = check
    return pos

def main_3_5(save=True):
    print("Load transcript data")
    df:pd.DataFrame = pd.read_pickle("Data/3_all_utters_impute.pkl")
    word_ids = df["word_id"].tolist()
    words = dict(zip(word_ids, df["word"].tolist()))
    utterances = dict(zip(word_ids, df["utterance_id"].tolist()))

    print("Tokenizing")
    tokens = {word_id: word_tokenize(word) for word_id, word in
words.items()}

    print("POS Tagging")

    utterance_tokens = []
    utterance_ids = []
    tagged_u = {}
    tagged = {}
```

```
for i, id in enumerate(word_ids):
    token = tokens[id]
    utterance = utterances[id]
    utterance_tokens.extend(token)
    utterance_ids.extend([id]*len(token))
    if id == word_ids[-1] or utterance != utterances[word_ids[i+1]]:
        utterance_tagged_u = nltk.pos_tag(utterance_tokens, tagset =
"universal")
        utterance_tagged = nltk.pos_tag(utterance_tokens)
        for id, pos_u in zip(utterance_ids, utterance_tagged_u):
            tagged_u[id] = tagged_u.get(id, []) + [pos_u]
        for id, pos in zip(utterance_ids, utterance_tagged):
            tagged[id] = tagged.get(id, []) + [pos]
        utterance_tokens = []
        utterance_ids = []
pos_u = {}
pos = {}
for id, tagged_tokens_u in tagged_u.items():
    pos_u[id] = pos_universal_resolution(tagged_tokens_u)
for id, tagged_tokens in tagged.items():
    pos[id] = pos_resolution(tagged_tokens)

print("POS Imputation")
df["pos o"] = df["pos"]
df["pos 2"] = df.apply(lambda x: pos[x["word_id"]], axis=1)
df["pos"] = df.apply(lambda x: x["pos o"] if pd.notna(x["pos o"])
else pos[x["word_id"]], axis=1)
df["pos u"] = df.apply(lambda x: pos_u[x["word_id"]], axis=1)

print("Preliminary scoring")
print("pos missing:", sum(pd.isna(df["pos o"])), "/", len(df["pos
o"]))
print("pos = pos 3:", sum(df["pos o"] == df["pos 2"]), "/",
sum(pd.notna(df["pos o"])))

if save:
    print("Saving files")
    df.to_excel("Data/3_5_all_utters_impute_pos.xlsx", "all")
    df.to_pickle("Data/3_5_all_utters_impute_pos.pkl")

# PART 3_7

def add_phoneme_count(df:pd.DataFrame):
    g2p = G2p()
    for i, row in df.iterrows():
        phonemes = g2p(row["word"])
        phoneme_count = sum(1 for ph in phonemes if ph != " ")
        syllable_count = sum(1 for ph in phonemes if ph[-1].isdigit())
        df.at[i, "phoneme count"] = phoneme_count
        df.at[i, "syllable count"] = syllable_count
    return df

def main_3_7():
```

```
df:pd.DataFrame =
pd.read_pickle("Data/3_5_all_utters_impute_pos.pkl")
df = add_phoneme_count(df)
df.to_excel("Data/3_7_all_utters_add_phonemes.xlsx", "all")
df.to_pickle("Data/3_7_all_utters_add_phonemes.pkl")

# PART 4

class subsetter:
    def __init__(self):
        self.original =
pd.read_pickle("Data/3_7_all_utters_add_phonemes.pkl")
    def no_filter(self, save=True, saveall=True):
        print("No filter")
        df = self.original.copy()
        if save:
            df.to_pickle("Data/4_0_all_utters.pkl")
            try:
                with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
                    df.to_excel(writer, sheet_name="0_all_utters")
            except FileNotFoundError:
                with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl') as writer:
                    df.to_excel(writer, sheet_name="0_all_utters")
            print("No filter complete")
            return df
    def filter_untranscribed(self, save=True, saveall=True):
        print("Filter untranscribed")
        df = self.no_filter(saveall, saveall)
        # Filter Untranscribed
        df_filter =
self.original.loc[~df["utterance_id"].isin(df.loc[df["error_untranscribed"]
]["utterance_id"])]
        df_removed =
self.original.loc[df["utterance_id"].isin(df.loc[df["error_untranscribed"]
]["utterance_id"])]
        if save:
            df_filter.to_pickle("Data/4_1_no_untranscribed.pkl")
            with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
                df_filter.to_excel(writer,
sheet_name="1_no_untranscribed")
            with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
                df_removed.to_excel(writer,
sheet_name="1_no_untranscribed_removed")
            return df_filter
    def filter_missing_word(self, save=True, saveall=True):
        print("Filter missing word")
        df = self.filter_untranscribed(saveall, saveall)
        # Filter missing word
        df_filter = df[(df["word"]!="x") & (df["word"]!="xxx") &
(df["word"] != "www") & (pd.notna(df["pos"])) & (pd.notna(df["word"]))]
```

```
df_removed = df[(df["word"]=="x") | (df["word"]=="xxx") |
(df["word"] == "www") | (pd.isna(df["pos"])) | (pd.isna(df["word"]))]
    if save:
        df_filter.to_pickle("Data/4_2_no_missing_words.pkl")
        with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
            df_filter.to_excel(writer,
sheet_name="2_no_missing_words")
            with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
                df_removed.to_excel(writer,
sheet_name="2_no_missing_words_removed")
        return df_filter
    def filter_non_ps_errors(self, save=True, saveall=True):
        print("Filter non ps errors")
        df = self.filter_missing_word(saveall, saveall)
        # Filter non ps errors
        df_filter = df[~((df["error_tag"]) & (~df["error_replacement"]))]
        df_removed = df[((df["error_tag"]) & (~df["error_replacement"]))]
        if save:
            df_filter.to_pickle("Data/4_3_only_ps_errors.pkl")
            with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
                df_filter.to_excel(writer, sheet_name="3_only_ps_errors")
            with pd.ExcelWriter("Data/4_all_filters.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
                df_removed.to_excel(writer,
sheet_name="3_only_ps_errors_removed")
        def create_spn_word_data(self, save=True, saveall=True,
with_imageability=False):
            print("Create spn word data")
            #df = self.filter_non_ps_errors(saveall, saveall)
            df = self.filter_missing_word(saveall, saveall)
            # Include numeric and categorical features
            numerics = ["location", "Age", "frequency 3",
                        "Aphasia duration", "syllable count"]
            if with_imageability:
                numerics = numerics + ["imageability 3"]
            categoricals = ["pos", "Fluency", "Apraxia", "Dysarthria",
"severity_very_severe",
                        "severity_severe", "severity_moderate",
"severity_mild"]
            cats_df = pd.get_dummies(df[categoricals])
            nums_df = df[numerics].copy()
            nums_df = nums_df.fillna(nums_df.mean())
            new_df_data = cats_df.join(nums_df)
            # Define Y
            error_presence_p = (df["error_tag_p"])
            error_presence_s = (df["error_tag_s"])
            error_presence_n = (df["error_tag_n"])
            error_presence = error_presence_p | error_presence_s |
error_presence_n
            error_presence = error_presence.copy()
            new_df_data["error"] = error_presence
```

```
new_df_data["error_p"] = error_presence_p
new_df_data["error_s"] = error_presence_s
new_df_data["error_n"] = error_presence_n
# Finalize and save
new_df_data = new_df_data.apply(pd.to_numeric)
if save:
    new_df_data.to_excel("Data/singles_spn_dataset.xlsx", "all")
    new_df_data.to_pickle("Data/singles_spn_dataset.pkl")
return new_df_data

def create_spn_window_data(self, window=[-1,0,1], padding=True,
error_feature=True, save=True, saveall=True, with_imageability=False):
    print("Create spn window data")
    #df = self.filter_non_ps_errors(saveall, saveall)
    df = self.filter_missing_word(saveall, saveall)
    # Include numeric and categorical features
    numerics = ["location", "Age", "frequency 3",
                "Aphasia duration", "syllable count"]
    if with_imageability:
        numerics = numerics + ["imageability 3"]
    numerics_person = ["Age", "Aphasia duration"]
    numerics_word = ["location", "frequency 3", "syllable count"]
    if with_imageability:
        numerics_word = numerics_word + ["imageability 3"]
    categoricals = ["pos", "Fluency", "Apraxia", "Dysarthria",
"severity_very_severe",
                "severity_severe", "severity_moderate",
"severity_mild"]
    categoricals_person = ["Fluency", "Apraxia", "Dysarthria",
"severity_very_severe", "severity_severe", "severity_moderate",
"severity_mild"]
    categoricals_word = ["pos"]
    cats_df_person = pd.get_dummies(df[categoricals_person])
    cats_df_word = pd.get_dummies(df[categoricals_word])
    cats_df = pd.get_dummies(df[categoricals])
    nums_df_person = df[numerics_person].copy()
    nums_df_person = nums_df_person.fillna(nums_df_person.mean())
    nums_df_word = df[numerics_word].copy()
    nums_df_word = nums_df_word.fillna(nums_df_word.mean())
    nums_df = df[numerics].copy()
    nums_df = nums_df.fillna(nums_df.mean())
    features_df_person = cats_df_person.join(nums_df_person)
    features_df_word = cats_df_word.join(nums_df_word)
    features_df = cats_df.join(nums_df)
    # Define Y
    error_presence_p = (df["error_tag_p"])
    error_presence_s = (df["error_tag_s"])
    error_presence_n = (df["error_tag_n"])
    error_presence = error_presence_p | error_presence_s |
error_presence_n
    error_presence = error_presence.copy()
    features_df["error"] = error_presence
    features_df_word["error"] = error_presence
    features_df["error_p"] = error_presence_p
    features_df_word["error_p"] = error_presence_p
```

```
features_df["error_s"] = error_presence_s
features_df_word["error_s"] = error_presence_s
features_df["error_n"] = error_presence_n
features_df_word["error_n"] = error_presence_n
# Get utterance index
utts_unique = df["utterance_id"].unique()
# Check uniques
print("Utterances: ", len(utts_unique))
print("Head: ", utts_unique[:10])
# Create windows
windows = []
for utt in utts_unique:
    utt_df_word = features_df_word.loc[df["utterance_id"]==utt]
    utt_df_word = utt_df_word.sort_values(by=["location"])
    utt_df_word = utt_df_word.reset_index(drop=True)
    utt_df_person =
features_df_person.loc[df["utterance_id"]==utt]
    utt_df_person = utt_df_person.reset_index(drop=True)
    if padding:
        centers = range(len(utt_df_word))
    else:
        centers = range(-min(window[0], 0), len(utt_df_word)-
max(window[-1], 0))
    len_row = len(utt_df_word.iloc[0,:])
    if utt % 100 == 0:
        print("Utterance: ", utt)
    for i in centers:
        window_dict = {"p": utt_df_person.iloc[i,:].copy()}
        for w in window:
            if i+w>=0 and i+w<len(utt_df_word):
                window_dict[str(w)] =
utt_df_word.iloc[i+w,:].copy()
            else:
                window_dict[str(w)] = pd.Series(np.zeros(len_row,
dtype=float), index=utt_df_word.columns)

        if w != 0 and not error_feature:
            # Drop error columns
            window_dict[str(w)] =
window_dict[str(w)].drop(["error", "error_p", "error_s", "error_n"])
        if 0 not in window:
            window_dict["0"] = utt_df_word.iloc[i,:][["error",
"error_p", "error_s", "error_n"]].copy()
        windows.append(window_dict)
# Create new dataframe from windows
new_df_data = pd.DataFrame()
flat_windows = []
for window_dict in windows:
    flat_windows.append(pd.concat(window_dict, axis=0))
new_df_data = pd.concat(flat_windows, axis=1).T
new_df_data.columns = ['_'.join(col).strip() for col in
new_df_data.columns.values]
# Swapping name of columns
```

```
new_df_data = new_df_data.rename(columns={"0_error": "error",
"0_error_p": "error_p", "0_error_s": "error_s", "0_error_n": "error_n"})
# Finalize and save
new_df_data = new_df_data.apply(pd.to_numeric)
img_string = "_img" if with_imageability else ""
ef_string = "_ef" if error_feature else ""
pad_string = "_pad" if padding else ""
if save:
    print(f"Saving to
Data/L{len(window)}_M{str(sum(window)/len(window)).replace('.', '-')}
{ef_string}{pad_string}{img_string}_window_spn_dataset")

new_df_data.to_excel(f"Data/L{len(window)}_M{str(sum(window)/len(window))
.replace('.', '-')}
{ef_string}{pad_string}{img_string}_window_spn_dataset.xlsx", "all")

new_df_data.to_pickle(f"Data/L{len(window)}_M{str(sum(window)/len(window))
.replace('.', '-')}
{ef_string}{pad_string}{img_string}_window_spn_dataset.pkl")
return new_df_data

def create_all_word():
    save = subsetter()
    save.create_spn_word_data(saveall=False, with_imageability=False)
    save.create_spn_word_data(saveall=True, with_imageability=True)

def create_all_window():
    for with_img in [True, False]:
        for pad, ef in [(True, True), (False, True), (True, False),
(False, False)]:
            # for pad, ef in [(True, True)]:
                for window in [[0], [-2,-1,0], [-1,0,1], [-4,-3,-2,-1,0], [-3,-
2,-1,0,1], [-2,-1,0,1,2], [-3,-2,-1,0,1,2,3]]:
                    # for window in [[-3,-2,-1]]:
                        # for window in [[-1,1], [-2,-1], [-4,-3,-2,-1], [-3,-2,-1,1], [-
2,-1,1,2], [-3,-2,-1,1,2,3]]:
                            save = subsetter()
                            save.create_spn_window_data(window=window, padding=pad,
error_feature=ef, saveall=False, with_imageability=with_img)

def main_4():
    create_all_word()
    create_all_window()

# MAIN

def main():
    to_do = [1, 2, 3, 3.5, 3.7, 4]
    # to_do = [4]
    print("Starting full pipeline")
    if 1 in to_do:
        print("Part 1 starting")
        main_1()
        print("Part 1 complete")
```

```
if 2 in to_do:
    print("Part 2 starting")
    main_2()
    print("Part 2 complete")
if 3 in to_do:
    print("Part 3 starting")
    main_3()
    print("Part 3 complete")
if 3.5 in to_do:
    print("Part 3_5 starting")
    main_3_5()
    print("Part 3_5 complete")
if 3.7 in to_do:
    print("Part 3_7 starting")
    main_3_7()
    print("Part 3_7 complete")
if 4 in to_do:
    print("Part 4 starting")
    main_4()
    print("Part 4 complete")
print("Pipeline complete")

if __name__ == "__main__":
    main()
```

**dataset.linear\_imputation\_models.py – This script creates imputation models to impute missing numeric features.**

```
import csv
import numpy as np
import pandas as pd

import gensim.downloader as api
from sklearn.linear_model import ElasticNetCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold

import pickle

def get_bird_db():
    img = {}
    aoa = {}
    freq = {}
    with open("Data/bird database.tsv", "r") as db:
        word_list = list(csv.DictReader(db, delimiter="\t"))
        for w in word_list:
            word = w["Word"].lower()
            imgval = w["new_Imageability"]
            aoaval = w["new_AoA"]
            freqval = w["Frequency"]
            img[word] = float(imgval) if imgval is not None and imgval !=
"." else None
            aoa[word] = float(aoaval) if aoaval is not None and aoaval !=
"." else None
            freq[word] = float(freqval) if freqval is not None and
freqval != "." else None
        return {"imageability 1":img, "age of acquisition 1":aoa, "frequency
1":freq}

def get_mrc2_db():
    img = {}
    aoa = {}
    freq = {}
    with open("Data/mrc2.dct", "r") as db:
        for line in db:
            a = {}
            s = line.split("|")
            clean = lambda a: int(a) if int(a) != 0 else None
            a["WORD"] = s[0][51:].lower()
            a["NLET"] = clean(s[0][0:2])
            a["NPHON"] = clean(s[0][2:3])
            a["NSYL"] = clean(s[0][4])
            a["K-F-FREQ"] = clean(s[0][5:10])
            a["T-L-FREQ"] = clean(s[0][15:21])
            a["BROWN-FREQ"] = clean(s[0][21:25])
            a["FAM"] = clean(s[0][25:28])
```

```
a["CONC"] = clean(s[0][28:31])
a["IMAG"] = clean(s[0][31:34])
a["AOA"] = clean(s[0][40:43])

word = a["WORD"]
img[word] = a["IMAG"]
aoa[word] = a["AOA"]
freq[word] = a["BROWN-FREQ"]
return {"imageability 2": img, "age of acquisition 2": aoa,
        "frequency 2": freq}

def load_word_vectors():
    wv = api.load('word2vec-google-news-300')
    # wv = api.load('conceptnet-numberbatch-17-06-300')
    # wv = api.load('glove-wiki-gigaword-50')
    return wv

def main(e, save=False):
    print("Loading word vectors")
    wv = load_word_vectors()
    print("Loading Bird database")
    db1 = get_bird_db()
    print("Loading MRC2 database")
    db2 = get_mrc2_db()
    db = db1 | db2
    for index, word in enumerate(wv.index_to_key):
        if index == 20:
            break
        print(f"word #{index}/{len(wv.index_to_key)} is {word}")
    dbboth = {}
    for feature_name, words in db.items():
        total = 0
        hasfeat = 0
        invec = 0
        both = 0
        hasboth = {}
        hasfeatdict = {}
        for word, value in words.items():
            total += 1
            if value is not None:
                hasfeat += 1
                hasfeatdict[word] = value
            if word in wv or word.title() in wv:
                invec += 1
            if (word in wv or word.title() in wv) and value is not None:
                both += 1
                if word in wv:
                    hasboth[word] = value
                else:
                    hasboth[word.title()] = value
        dbboth[feature_name] = hasboth
    print(feature_name)
    print(f"Feature present in: {hasfeat}/{total}")
    print(f"Word in wordvec: {invec}/{total}")
```

```
    print(f"Both: {both}/{total}")
train_scores_db = {}
test_scores_db = {}
models_db = {}
for feature_name, words in dbboth.items():
    X = np.stack([wv[word] for word in words.keys()])
    y = np.stack([value for value in words.values()])
    print("Min, Mean, Max")
    print(np.min(y), np.mean(y), np.max(y))
    reps = 1
    train_scores = []
    test_scores = []
    models = []
    for i in range(reps):
        kf = KFold(n_splits = 5, shuffle = True)
        for train_i, test_i in kf.split(X):
            X_train, X_test = X[train_i], X[test_i]
            y_train, y_test = y[train_i], y[test_i]
            model = ElasticNetCV()
            model.fit(X_train, y_train)
            train_score = model.score(X_train, y_train)
            test_score = model.score(X_test, y_test)
            print(f"Train score: {train_score}")
            print(f"Test score: {test_score}")
            train_scores.append(train_score)
            test_scores.append(test_score)
            models.append(model)
    train_scores_db[feature_name] = train_scores
    test_scores_db[feature_name] = test_scores
    models_db[feature_name] = models
for feature_name in train_scores_db.keys():
    print(f"{feature_name} Scores:")
    print(f"Training: {np.mean(train_scores_db[feature_name])}")
    print(f"Testing {np.mean(test_scores_db[feature_name])}")
    best_model =
models_db[feature_name][np.argmax(test_scores_db[feature_name])]
    print(f"Best Test: {np.max(test_scores_db[feature_name])}")
    if save:
        with open(f"Models/{feature_name}_lr_model.pkl", 'wb') as
file:
            pickle.dump(best_model, file)
    excel = {f"train_{k}": v for k, v in train_scores_db.items()} |
{f"test_{k}": v for k, v in test_scores_db.items()}
    if save:
        with pd.ExcelWriter("Data/word_vec_caps_lr.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
            pd.DataFrame(excel).to_excel(writer, sheet_name=f"used")
        # with pd.ExcelWriter("Data/word_vec_caps_glove_lr.xlsx",
engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
        #     pd.DataFrame(excel).to_excel(writer, sheet_name=f"_")

if __name__ == "__main__":
    main(0.0, save=False)
```

**train\_test\_models.py – This script trains and tests models using parsed feature data.**

```
from copy import deepcopy
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import RocCurveDisplay
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import MinMaxScaler
import os
import matplotlib.pyplot as plt
import argparse
import pickle

import pandas as pd

class SVM_Factory():
    def __init__(self):
        self.scale = True

    def train(self, x, y):
        model = SVC()
        hist = model.fit(x, y)
        return model, hist

    def name_front(self):
        return "svm"

class Logistic_Regression_Factory():
    def __init__(self):
        self.scale = True

    def train(self, x, y):
        model = LogisticRegression(penalty=None)
        hist = model.fit(x, y)
        return model, hist

    def name_front(self):
        return "lr"

class Random_Forest_Factory():
    def __init__(self, ntrees=100, maxdepth=-1, balanced=False):
        self.scale = False
        self.ntrees = ntrees
        self.maxdepth = None if maxdepth == -1 else maxdepth
        self.balanced = balanced
        self.class_weight = "balanced" if balanced else None

    def train(self, x, y):
```

```
        model = RandomForestClassifier(n_estimators=self.ntrees,
max_depth=self.maxdepth, class_weight=self.class_weight)
        hist = model.fit(x, y)
        return model, hist

    def name_front(self):
        name = "rf"
        if self.balanced:
            name += "_bal"
        if self.maxdepth is not None:
            name += "_md" + str(self.maxdepth)
        return name

class Decision_Tree_Factory():
    def __init__(self, maxdepth=-1, ccp=-1, balanced=False):
        self.scale = False
        self.maxdepth = None if maxdepth == -1 else maxdepth
        self.ccp = None if ccp == -1 else ccp
        self.balanced = balanced
        self.class_weight = "balanced" if balanced else None

    def train(self, x, y):
        if self.ccp is not None:
            model = DecisionTreeClassifier(max_depth=self.maxdepth,
ccp_alpha = self.ccp, class_weight=self.class_weight)
        else:
            model = DecisionTreeClassifier(max_depth=self.maxdepth,
class_weight=self.class_weight)
        hist = model.fit(x, y)
        return model, hist

    def name_front(self):
        name = "dt"
        if self.balanced:
            name += "_bal"
        if self.maxdepth is not None:
            name += "_md" + str(self.maxdepth)
        if self.ccp is not None:
            name += "_ccp" + str(self.ccp)
        return name

class Model_Test():
    def __init__(self, data_source="Data/L3_M0-
0_pad_window_spn_dataset.pkl", model_factory=None, do_train=True,
do_test=True, plot=0, reps=100, do_img=False):
        self.data_source = data_source
        self.data = None
        self.do_train = do_train
        self.do_test = do_test
        self.plot = plot
        self.model_factory = model_factory
        self.reps = reps
        self.do_img = do_img
        self.stats = {
```

```
        "neg_test_scores": [],
        "neg_test_counts": [],
        "pos_s_test_scores": [],
        "pos_s_test_counts": [],
        "pos_p_test_scores": [],
        "pos_p_test_counts": [],
        "pos_n_test_scores": [],
        "pos_n_test_counts": [],
        "neg_train_scores": [],
        "neg_train_counts": [],
        "pos_s_train_scores": [],
        "pos_s_train_counts": [],
        "pos_p_train_scores": [],
        "pos_p_train_counts": [],
        "pos_n_train_scores": [],
        "pos_n_train_counts": [],
        "auc": [],
        "feature_importances": []
    }

    def get_data(self, scale):
        if self.data is None:
            data = pd.read_pickle(self.data_source)
            data = data.loc[:, ~data.columns.isin([
                "age of acquisition 3", "-4_age of acquisition 3", "-3_age of acquisition 3", "-2_age of acquisition 3", "-1_age of acquisition 3", "0_age of acquisition 3", "1_age of acquisition 3", "2_age of acquisition 3", "3_age of acquisition 3"
            ])]
            X = data.loc[:, ~data.columns.isin(["error", "error_p", "error_s", "error_n"])]
            X = X.to_numpy()
            y = data["error"].to_numpy()
            p = data["error_p"].to_numpy()
            s = data["error_s"].to_numpy()
            n = data["error_n"].to_numpy()
            self.data = {"data":data, "X":X, "y":y, "p":p, "s":s, "n":n}
        return self.data

    def print_data_info(self):
        data = self.get_data()
        X = data["X"]
        y = data["y"]
        p = data["p"]
        s = data["s"]
        n = data["n"]

        print("P Error count: ", np.sum(p))
        print("S Error count: ", np.sum(s))
        print("N Error count: ", np.sum(n))
        print("Percent Aphasias:" + str(np.mean(y)))
        print("Total number of samples: " + str(len(y)))
        print()
```

```
def special_name(self):
    # split on / or \ folder indicator, then split on file extension
    special_name = self.data_source.split("/")[-1].split("\\")[-
1].split(".")[0] + "/"
    special_name += self.model_factory.name_front()
    return special_name

def run_kfold(self):
    if self.do_img > 0:
        import dtreeviz
    data = self.get_data(self.model_factory.scale)
    X = data["X"]
    y = data["y"]
    p = data["p"]
    s = data["s"]
    n = data["n"]
    stats = self.stats
    i = 0
    self.reps = 100
    best = {"auc": 0, "model": None, "hist": None, "special_name":
""}

    for rep in range(self.reps):
        if rep % 20 == 0:
            print("Rep: ", rep, "/", self.reps)
        kf = KFold(n_splits=5, shuffle=True)
        for train_i, test_i in kf.split(X):
            X_train, X_test = X[train_i], X[test_i]
            y_train, y_test = y[train_i], y[test_i]
            model, hist = self.model_factory.train(X_train, y_train)

            X_p_pos = X_test[p[test_i]]
            X_s_pos = X_test[s[test_i]]
            X_n_pos = X_test[n[test_i]]
            X_neg = X_test[y_test==0]

            neg_score = model.score(X_neg, y_test[y_test==0])
            pos_p_score = model.score(X_p_pos, y_test[p[test_i]])
            pos_s_score = model.score(X_s_pos, y_test[s[test_i]])
            pos_n_score = model.score(X_n_pos, y_test[n[test_i]])

            neg_count = np.sum(y_test==0)
            pos_p_count = np.sum(p[test_i])
            pos_s_count = np.sum(s[test_i])
            pos_n_count = np.sum(n[test_i])
            stats["neg_test_scores"].append(neg_score)
            stats["pos_p_test_scores"].append(pos_p_score)
            stats["pos_s_test_scores"].append(pos_s_score)
            stats["pos_n_test_scores"].append(pos_n_score)
            stats["neg_test_counts"].append(neg_count)
            stats["pos_p_test_counts"].append(pos_p_count)
            stats["pos_s_test_counts"].append(pos_s_count)
            stats["pos_n_test_counts"].append(pos_n_count)
            neg_score = model.score(X_train[y_train==0],
y_train[y_train==0])
```



```
#         class_names=["no", "yes"]).view()

#
viz.save(f"Out/snptestimg/decision_tree_md{special_name}_test.svg")
i += 1

errors = pd.DataFrame({k: v for k, v in stats.items() if k !=
"feature_importances"})
errors.loc["mean"] = errors.mean()
# print("Negative Mean :",
str(np.mean(stats["neg_test_scores"])))
# print("Positive Mean p:",
str(np.mean(stats["pos_p_test_scores"])))
# print("Positive Mean s:",
str(np.mean(stats["pos_s_test_scores"])))
# print("Positive Mean n:",
str(np.mean(stats["pos_n_test_scores"])))
importances = pd.DataFrame({feature: importance for feature,
importance in zip(list(data["data"].columns),
list(np.transpose(np.array(stats["feature_importances"])))))
try:
    importances_mean = importances.mean()
except TypeError as err:
    importances_mean = importances

# print("Features: ", str(list(data["data"].columns)))
# print("Feature importance: ",
str(np.mean(stats["feature_importances"], axis=0)))
# print("alphas:", model.cost_complexity_pruning_path(X_train,
y_train)["ccp_alphas"])
special_name = self.special_name()
print(special_name.rsplit("/", 1)[0])
print(special_name.rsplit("/", 1)[1])
os.makedirs(f"Out/{special_name.rsplit('/', 1)[0]}",
exist_ok=True)
with pd.ExcelWriter(f"Out/{special_name}.xlsx") as writer:
    errors.to_excel(writer, "errors")
    importances.to_excel(writer, "feature_importance")
    importances_mean.to_excel(writer, "feature_importance_mean")
    os.makedirs(f"Models/{special_name.rsplit('/', 1)[0]}",
exist_ok=True)
    with open(f"Models/{best['special_name']}.pkl", 'wb') as file:
        pickle.dump(best["model"], file)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("model", type=str, choices=["rf", "dt", "lr",
"svm"])
    parser.add_argument("-src", "--source", type=str,
default="Data/L3_M0-0_pad_window_spn_dataset.pkl")
    parser.add_argument("-spn", "--spn", action="store_true",
default=True)
    parser.add_argument("-sp", "--sp", action="store_true",
default=False)
```

```
parser.add_argument("-t", "--train", action="store_true")
parser.add_argument("-md", "--maxdepth", type=int, default=-1)
parser.add_argument("-mnl", "--minloop", type=int, default=-1)
parser.add_argument("-mxl", "--maxloop", type=int, default=-1)
parser.add_argument("-of", "--onlyfeature", action="store_true")
parser.add_argument("-c", "--ccp", type=float, default=-1)
parser.add_argument("-img", "--image", action="store_true")
parser.add_argument("-b", "--balanced", action="store_true")
parser.add_argument("-plt", "--plot", type=int, default=0)
args = parser.parse_args()
if args.model == "rf":
    model_factory = Random_Forest_Factory(maxdepth=args.maxdepth,
balanced=args.balanced)
    elif args.model == "lr":
        model_factory = Logistic_Regression_Factory()
    elif args.model == "dt":
        model_factory = Decision_Tree_Factory(maxdepth=args.maxdepth,
ccp=args.ccp, balanced=args.balanced)
    elif args.model == "svm":
        model_factory = SVM_Factory()
    else:
        print("Model not found")
        return
    model_test = Model_Test(data_source=args.source,
model_factory=model_factory, do_train=args.train, plot=args.plot)
    model_test.run_kfold()

# if not args.sp:
#     if args.train:
#         if args.spn:
#             train_new(maxdepth=args.maxdepth, ccp=args.ccp,
image=args.image, balanced=args.balanced, plot=args.plot)
#         else:
#             # Train and test model
#             print("That function may be missing")
#     else:
#         if args.train:
#             if args.onlyfeature:
#                 for i in range(48):
#                     train_sp_word_data_only_feature(feature_id=i,
maxdepth=2, image=args.image, balanced=args.balanced, plot=args.plot)
#                 elif args.minloop != -1:
#                     for i in range(args.minloop, args.maxloop):
#                         train_sp_word_data(maxdepth=i, image=args.image,
balanced=args.balanced, plot=args.plot)
#                 else:
#                     train_sp_word_data(maxdepth=args.maxdepth,
ccp=args.ccp, image=args.image, balanced=args.balanced, plot=args.plot)

def train_new(reps = 100, maxdepth=-1, ccp=-1, image=False,
balanced=False, plot=False):
    maxdepth = None if maxdepth == -1 else maxdepth
    class_weight = "balanced" if balanced else None
```

```
data = pd.read_pickle("Data/singles_spn_dataset.pkl")
X = data.loc[:, ~data.columns.isin(["error", "error_p", "error_s",
"error_n"])]].to_numpy()
y = data["error"].to_numpy()
p = data["error_p"].to_numpy()
s = data["error_s"].to_numpy()
n = data["error_n"].to_numpy()
# train_test_split(x, y, test_size=0.2)
print(X)
print()
print(y)
r = {
    "neg_test_scores": [],
    "neg_test_counts": [],
    "pos_s_test_scores": [],
    "pos_s_test_counts": [],
    "pos_p_test_scores": [],
    "pos_p_test_counts": [],
    "pos_n_test_scores": [],
    "pos_n_test_counts": [],
    "neg_train_scores": [],
    "neg_train_counts": [],
    "pos_s_train_scores": [],
    "pos_s_train_counts": [],
    "pos_p_train_scores": [],
    "pos_p_train_counts": [],
    "pos_n_train_scores": [],
    "pos_n_train_counts": []
}
feature_importances = []
for i in range(reps):
    kf = KFold(n_splits=5, shuffle=True)
    # print("Percent Aphasias:" + str(np.mean(y)))
    # scores = []
    # for train_i, test_i in kf.split(X):
    #     X_train, X_test = X[train_i], X[test_i]
    #     y_train, y_test = y[train_i], y[test_i]
    #     model = RandomForestClassifier(n_estimators=100)
    #     model.fit(X_train, y_train)
    #     score = model.score(X_test, y_test)
    #     print(score)
    #     scores.append(score)
    # print("Mean: " + str(np.mean(scores)))

print("P Error count: ", np.sum(p))
print("S Error count: ", np.sum(s))
print("N Error count: ", np.sum(n))
print("With weighted samples")
for train_i, test_i in kf.split(X):
    X_train, X_test = X[train_i], X[test_i]
    y_train, y_test = y[train_i], y[test_i]
    print("P Error count train: ", np.sum(s[train_i]))
    print("S Error count train: ", np.sum(p[train_i]))
    print("N error count train: ", np.sum(n[train_i]))
```

```
        if ccp != -1:
            model = DecisionTreeClassifier(max_depth=maxdepth,
            ccp_alpha=ccp, class_weight=class_weight)
        else:
            model = DecisionTreeClassifier(max_depth=maxdepth)
    model.fit(X_train, y_train)
    X_p_pos = X_test[p[test_i]]
    X_s_pos = X_test[s[test_i]]
    X_n_pos = X_test[n[test_i]]
    X_neg = X_test[y_test==0]
    neg_score = model.score(X_neg, y_test[y_test==0])
    pos_p_score = model.score(X_p_pos, y_test[p[test_i]])
    pos_s_score = model.score(X_s_pos, y_test[s[test_i]])
    pos_n_score = model.score(X_n_pos, y_test[n[test_i]])
    neg_count = np.sum(y_test==0)
    pos_p_count = np.sum(p[test_i])
    pos_s_count = np.sum(s[test_i])
    pos_n_count = np.sum(n[test_i])
    print("True negative   : ", str(neg_score), "from", neg_count)
    print("True positive p: ", str(pos_p_score), "from",
pos_p_count)
    print("True positive s: ", str(pos_s_score), "from",
pos_s_count)
    print("True positive n: ", str(pos_n_score), "from",
pos_n_count)
    r["neg_test_scores"].append(neg_score)
    r["pos_p_test_scores"].append(pos_p_score)
    r["pos_s_test_scores"].append(pos_s_score)
    r["pos_n_test_scores"].append(pos_n_score)
    r["neg_test_counts"].append(neg_count)
    r["pos_p_test_counts"].append(pos_p_count)
    r["pos_s_test_counts"].append(pos_s_count)
    r["pos_n_test_counts"].append(pos_n_count)
    neg_score = model.score(X_train[y_train==0],
y_train[y_train==0])
    pos_p_score = model.score(X_train[p[train_i]],
y_train[p[train_i]])
    pos_s_score = model.score(X_train[s[train_i]],
y_train[s[train_i]])
    pos_n_score = model.score(X_train[n[train_i]],
y_train[n[train_i]])
    neg_count = np.sum(y_train==0)
    pos_p_count = np.sum(p[train_i])
    pos_s_count = np.sum(s[train_i])
    pos_n_count = np.sum(n[train_i])
    print("Training True negative   : ", str(neg_score), "from",
np.sum(y_train==0))
    print("Training True positive p: ", str(pos_p_score), "from",
np.sum(p[train_i]))
    print("Training True positive s: ", str(pos_s_score), "from",
np.sum(s[train_i]))
    print("Training True positive n: ", str(pos_n_score), "from",
np.sum(n[train_i]))
    r["neg_train_scores"].append(neg_score)
```

```
r["pos_p_train_scores"].append(pos_p_score)
r["pos_s_train_scores"].append(pos_s_score)
r["pos_n_train_scores"].append(pos_n_score)
r["neg_train_counts"].append(neg_count)
r["pos_p_train_counts"].append(pos_p_count)
r["pos_s_train_counts"].append(pos_s_count)
r["pos_n_train_counts"].append(pos_n_count)
feature_importances.append(model.feature_importances_)

errors = pd.DataFrame(r)
print("Negative Mean  :", str(np.mean(r["neg_test_scores"])))
print("Positive Mean p:", str(np.mean(r["pos_p_test_scores"])))
print("Positive Mean s:", str(np.mean(r["pos_s_test_scores"])))
print("Positive Mean n:", str(np.mean(r["pos_n_test_scores"])))
importances = pd.DataFrame({feature: importance for feature,
importance in zip(list(data.columns),
list(np.transpose(np.array(feature_importances))))})
importances_mean = importances.mean()
print("Features: ", str(list(data.columns)))
print("Feature importance: ", str(np.mean(feature_importances,
axis=0)))
print("alphas:", model.cost_complexity_pruning_path(X_train,
y_train)["ccp_alphas"])
if plot:
    rocplot = RocCurveDisplay.from_estimator(model, X_test, y_test)
    plt.show()
special_name = "_"
if balanced:
    special_name += "bal_"
if ccp != -1:
    special_name += "ccp_" + str(ccp) + "_"
if maxdepth is not None:
    special_name += str(maxdepth)
with
pd.ExcelWriter(f"Out/snpxlsx/spn_single_model_dt{special_name}.xlsx") as
writer:
    errors.to_excel(writer, "errors")
    importances.to_excel(writer, "feature_importance")
    importances_mean.to_excel(writer, "feature_importance_mean")
if image:
    import dtreeviz

    viz = dtreeviz.model(model, X_train, y_train,
        target_name="error",
        feature_names=list(data.columns),
        class_names=["no", "yes"]).view()

viz.save(f"Out/snptraining/decision_tree_md{special_name}_train.svg")

viz = dtreeviz.model(model, X_test, y_test,
    target_name="error",
    feature_names=list(data.columns),
    class_names=["no", "yes"]).view()
```

```
viz.save(f"Out/snpTesting/decision_tree_md{special_name}_test.svg")

def train_sp_word_data_only_feature(reps = 1, maxdepth=-1, feature_id=-1,
image=False, balanced=False, plot=False):
    maxdepth = None if maxdepth == -1 else maxdepth
    class_weight = "balanced" if balanced else None

    data = pd.read_pickle("Data/singles_sp_dataset.pkl")
    used_feature = pd.DataFrame(data.loc[:, ~data.columns.isin(["error",
"error_p", "error_s"])]).iloc[:, feature_id])
    X = used_feature.to_numpy()
    y = data["error"].to_numpy()
    p = data["error_p"].to_numpy()
    s = data["error_s"].to_numpy()
    # train_test_split(x, y, test_size=0.2)
    print(X)
    print()
    print(y)
    r = {
        "neg_test_scores": [],
        "neg_test_counts": [],
        "pos_s_test_scores": [],
        "pos_s_test_counts": [],
        "pos_p_test_scores": [],
        "pos_p_test_counts": [],
        "neg_train_scores": [],
        "neg_train_counts": [],
        "pos_s_train_scores": [],
        "pos_s_train_counts": [],
        "pos_p_train_scores": [],
        "pos_p_train_counts": [],
    }
    # feature_importances = []
    for i in range(reps):
        kf = KFold(n_splits=5, shuffle=True)
        # print("Percent Aphasias:" + str(np.mean(y)))
        # scores = []
        # for train_i, test_i in kf.split(X):
        #     X_train, X_test = X[train_i], X[test_i]
        #     y_train, y_test = y[train_i], y[test_i]
        #     model = RandomForestClassifier(n_estimators=100)
        #     model.fit(X_train, y_train)
        #     score = model.score(X_test, y_test)
        #     print(score)
        #     scores.append(score)
        # print("Mean: " + str(np.mean(scores)))

    print("P Error count: ", np.sum(p))
    print("S Error count: ", np.sum(s))
    print("With weighted samples")
    for train_i, test_i in kf.split(X):
        X_train, X_test = X[train_i], X[test_i]
```

```
y_train, y_test = y[train_i], y[test_i]
print("P Error count train: ", np.sum(s[train_i]))
print("S Error count train: ", np.sum(p[train_i]))
model = DecisionTreeClassifier(max_depth=maxdepth,
class_weight=class_weight)
model.fit(X_train, y_train)
X_p_pos = X_test[p[test_i]]
X_s_pos = X_test[s[test_i]]
X_neg = X_test[y_test==0]
neg_score = model.score(X_neg, y_test[y_test==0])
pos_p_score = model.score(X_p_pos, y_test[p[test_i]])
pos_s_score = model.score(X_s_pos, y_test[s[test_i]])
neg_count = np.sum(y_test==0)
pos_p_count = np.sum(p[test_i])
pos_s_count = np.sum(s[test_i])
print("True negative : ", str(neg_score), "from", neg_count)
print("True positive p: ", str(pos_p_score), "from",
pos_p_count)
print("True positive s: ", str(pos_s_score), "from",
pos_s_count)
r["neg_test_scores"].append(neg_score)
r["pos_p_test_scores"].append(pos_p_score)
r["pos_s_test_scores"].append(pos_s_score)
r["neg_test_counts"].append(neg_count)
r["pos_p_test_counts"].append(pos_p_count)
r["pos_s_test_counts"].append(pos_s_count)
neg_score = model.score(X_train[y_train==0],
y_train[y_train==0])
pos_p_score = model.score(X_train[p[train_i]],
y_train[p[train_i]])
pos_s_score = model.score(X_train[s[train_i]],
y_train[s[train_i]])
neg_count = np.sum(y_train==0)
pos_p_count = np.sum(p[train_i])
pos_s_count = np.sum(s[train_i])
print("Training True negative : ", str(neg_score), "from",
np.sum(y_train==0))
print("Training True positive p: ", str(pos_p_score), "from",
np.sum(p[train_i]))
print("Training True positive s: ", str(pos_s_score), "from",
np.sum(s[train_i]))
r["neg_train_scores"].append(neg_score)
r["pos_p_train_scores"].append(pos_p_score)
r["pos_s_train_scores"].append(pos_s_score)
r["neg_train_counts"].append(neg_count)
r["pos_p_train_counts"].append(pos_p_count)
r["pos_s_train_counts"].append(pos_s_count)
# feature_importances.append(model.feature_importances_)
errors = pd.DataFrame(r)
print("Negative Mean : ", str(np.mean(r["neg_test_scores"])))
print("Positive Mean p: ", str(np.mean(r["pos_p_test_scores"])))
print("Positive Mean s: ", str(np.mean(r["pos_s_test_scores"])))
```

```
#importances = pd.DataFrame({feature: importance for feature,
importance in zip(list(data.columns),
list(np.transpose(np.array(feature_importances))))})
#importances_mean = importances.mean()
# print("Features: ", str(list(used_feature.columns)))
# print("Feature importance: ", str(np.mean(feature_importances,
axis=0)))
if plot:
    rocplot = RocCurveDisplay.from_estimator(model, X_test, y_test)
    plt.show()
special_name = "_"
if balanced:
    special_name += "bal_"
if maxdepth is not None:
    special_name += str(maxdepth) + "_"
special_name += str(list(used_feature.columns)[0])

with pd.ExcelWriter(f"Out/s_p_single_model_dt_{special_name}.xlsx")
as writer:
    errors.to_excel(writer, "errors")
    # importances.to_excel(writer, "feature_importance")
    # importances_mean.to_excel(writer, "feature_importance_mean")

if image:
    import dtreeviz

    viz = dtreeviz.model(model, X_train, y_train,
        target_name="error",
        feature_names=list(used_feature.columns),
        class_names=["no", "yes"]).view()

viz.save(f"Out/decision_tree_md_{list(used_feature.columns)[0]}_train.svg")

viz = dtreeviz.model(model, X_test, y_test,
    target_name="error",
    feature_names=list(used_feature.columns),
    class_names=["no", "yes"]).view()

viz.save(f"Out/decision_tree_md_{list(used_feature.columns)[0]}_test.svg")

def train_sp_word_data(reps = 1, maxdepth=-1, ccp=-1, image=False,
balanced=False, plot=False):
    maxdepth = None if maxdepth == -1 else maxdepth
    class_weight = "balanced" if balanced else None

    data = pd.read_pickle("Data/singles_sp_dataset.pkl")
    X = data.loc[:, ~data.columns.isin(["error", "error_p",
"error_s"])]
    y = data["error"].to_numpy()
    p = data["error_p"].to_numpy()
```

```
s = data["error_s"].to_numpy()
# train_test_split(x, y, test_size=0.2)
print(X)
print()
print(y)
r = {
    "neg_test_scores": [],
    "neg_test_counts": [],
    "pos_s_test_scores": [],
    "pos_s_test_counts": [],
    "pos_p_test_scores": [],
    "pos_p_test_counts": [],
    "neg_train_scores": [],
    "neg_train_counts": [],
    "pos_s_train_scores": [],
    "pos_s_train_counts": [],
    "pos_p_train_scores": [],
    "pos_p_train_counts": [],
}
feature_importances = []
for i in range(reps):
    kf = KFold(n_splits=5, shuffle=True)
    # print("Percent Aphasias:" + str(np.mean(y)))
    # scores = []
    # for train_i, test_i in kf.split(X):
    #     X_train, X_test = X[train_i], X[test_i]
    #     y_train, y_test = y[train_i], y[test_i]
    #     model = RandomForestClassifier(n_estimators=100)
    #     model.fit(X_train, y_train)
    #     score = model.score(X_test, y_test)
    #     print(score)
    #     scores.append(score)
    # print("Mean: " + str(np.mean(scores)))

    print("P Error count: ", np.sum(p))
    print("S Error count: ", np.sum(s))
    print("With weighted samples")
    for train_i, test_i in kf.split(X):
        X_train, X_test = X[train_i], X[test_i]
        y_train, y_test = y[train_i], y[test_i]
        print("P Error count train: ", np.sum(s[train_i]))
        print("S Error count train: ", np.sum(p[train_i]))
        if ccp != -1:
            model = DecisionTreeClassifier(max_depth=maxdepth,
            ccp_alpha=ccp, class_weight=class_weight)
        else:
            model = DecisionTreeClassifier(max_depth=maxdepth)
        model.fit(X_train, y_train)
        X_p_pos = X_test[p[test_i]]
        X_s_pos = X_test[s[test_i]]
        X_neg = X_test[y_test==0]
        neg_score = model.score(X_neg, y_test[y_test==0])
        pos_p_score = model.score(X_p_pos, y_test[p[test_i]])
        pos_s_score = model.score(X_s_pos, y_test[s[test_i]])
```

```
neg_count = np.sum(y_test==0)
pos_p_count = np.sum(p[test_i])
pos_s_count = np.sum(s[test_i])
print("True negative : ", str(neg_score), "from", neg_count)
print("True positive p: ", str(pos_p_score), "from",
pos_p_count)
print("True positive s: ", str(pos_s_score), "from",
pos_s_count)
r["neg_test_scores"].append(neg_score)
r["pos_p_test_scores"].append(pos_p_score)
r["pos_s_test_scores"].append(pos_s_score)
r["neg_test_counts"].append(neg_count)
r["pos_p_test_counts"].append(pos_p_count)
r["pos_s_test_counts"].append(pos_s_count)
neg_score = model.score(X_train[y_train==0],
y_train[y_train==0])
pos_p_score = model.score(X_train[p[train_i]],
y_train[p[train_i]])
pos_s_score = model.score(X_train[s[train_i]],
y_train[s[train_i]])
neg_count = np.sum(y_train==0)
pos_p_count = np.sum(p[train_i])
pos_s_count = np.sum(s[train_i])
print("Training True negative : ", str(neg_score), "from",
np.sum(y_train==0))
print("Training True positive p: ", str(pos_p_score), "from",
np.sum(p[train_i]))
print("Training True positive s: ", str(pos_s_score), "from",
np.sum(s[train_i]))
r["neg_train_scores"].append(neg_score)
r["pos_p_train_scores"].append(pos_p_score)
r["pos_s_train_scores"].append(pos_s_score)
r["neg_train_counts"].append(neg_count)
r["pos_p_train_counts"].append(pos_p_count)
r["pos_s_train_counts"].append(pos_s_count)
feature_importances.append(model.feature_importances_)

errors = pd.DataFrame(r)
print("Negative Mean :", str(np.mean(r["neg_test_scores"])))
print("Positive Mean p:", str(np.mean(r["pos_p_test_scores"])))
print("Positive Mean s:", str(np.mean(r["pos_s_test_scores"])))
importances = pd.DataFrame({feature: importance for feature,
importance in zip(list(data.columns),
list(np.transpose(np.array(feature_importances))))})
importances_mean = importances.mean()
print("Features: ", str(list(data.columns)))
print("Feature importance: ", str(np.mean(feature_importances,
axis=0)))
print("alphas:", model.cost_complexity_pruning_path(X_train,
y_train)["ccp_alphas"])
if plot:
    rocplot = RocCurveDisplay.from_estimator(model, X_test, y_test)
    plt.show()
special_name = "_"
```

```
if balanced:
    special_name += "bal_"
if ccp != -1:
    special_name += "ccp_" + str(ccp) + "_"
if maxdepth is not None:
    special_name += str(maxdepth)
with pd.ExcelWriter(f"Out/s_p_single_model_dt{special_name}.xlsx") as
writer:
    errors.to_excel(writer, "errors")
    importances.to_excel(writer, "feature_importance")
    importances_mean.to_excel(writer, "feature_importance_mean")
if image:
    import dtreeviz

    viz = dtreeviz.model(model, X_train, y_train,
                        target_name="error",
                        feature_names=list(data.columns),
                        class_names=["no", "yes"]).view()

    viz.save(f"Out/decision_tree_md{special_name}_train.svg")

    viz = dtreeviz.model(model, X_test, y_test,
                        target_name="error",
                        feature_names=list(data.columns),
                        class_names=["no", "yes"]).view()

    viz.save(f"Out/decision_tree_md{special_name}_test.svg")

if __name__ == "__main__":
    main()
```