```python
import pandas as pd
import numpy as np
import gspread
from google.colab import auth

from google.auth import default
from gspread_dataframe import get_as_dataframe
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')
!pip install wordfreq
from wordfreq import zipf_frequency
from sklearn.model_selection import StratifiedKFold, cross_validate
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_curve, auc, precision_recall_curve, average_precision_score
import seaborn as sns
from scipy.stats import fisher_exact
from statsmodels.stats.multitest import multipletests
```

```python
auth.authenticate_user()
creds, _ = default()
gc = gspread.authorize(creds)
from google.colab import drive
drive.mount('/content/drive')
```

```python
# Reading files
words = gc.open_by_url('https://docs.google.com/spreadsheets/d/1mDg1HuTNgquVK4a9T5Nu--21W0NJzsSPASMfl6A1gCo/edit?gid=0#gid=0')
pre = gc.open_by_url('https://docs.google.com/spreadsheets/d/1EvC0PwMzbipWXdxIbCXc6urVQZjDJV4oRNRG68WaWnQ/edit?resourcekey=&gid=974437907#gid=974437907')
post = gc.open_by_url('https://docs.google.com/spreadsheets/d/1rl_UsiBDM7flcRjVeB4HtysQkVA1M3fwJmDx7Ct1Cp0/edit?resourcekey=&gid=1754048364#gid=1754048364')

wordsheet = words.worksheet('Sheet1')
presheet = pre.worksheet('Sheet1')
postsheet = post.worksheet('Sheet1')

word = get_as_dataframe(wordsheet)
pre = get_as_dataframe(presheet)
post = get_as_dataframe(postsheet)

url = "/content/drive/MyDrive/Colab Notebooks/challenging_words_zipf.xlsx"
word_dataset = pd.read_excel(url)

medical_dys = gc.open_by_url('https://docs.google.com/spreadsheets/d/13W3FAc_f1tfZU3SXAKlBl3cyKoqPIWtHNJh3J1jm3t8/edit?gid=1533757235#gid=1533757235')
self_dys = gc.open_by_url('https://docs.google.com/spreadsheets/d/1Emyuq_kbaRgTJEDVRqF8dT2HA-L4ZsIs8SSzzuX0U7o/edit?gid=1428308089#gid=1428308089')

medsheet = medical_dys.worksheet('Sheet1')
selfsheet = self_dys.worksheet('Sheet1')

medical_dyslexic = get_as_dataframe(medsheet)
self_dyslexic = get_as_dataframe(selfsheet)
```

```
# add a dyslexia score based on the screener to the word dataset

pre = pre.rename(columns={'Please enter your Prolific ID:':'Prolific ID'})
pre = pre.drop_duplicates(subset='Prolific ID')
duplicate_ids = pre['Prolific ID'][pre['Prolific ID'].duplicated()]

pre['dys-score'] = ((pre.iloc[:,13:18]=='Yes').sum(axis=1))*3 + ((pre.iloc[:,18:24]=='Yes').sum(axis=1))*2 + ((pre.iloc[:,24:25]=='Yes').sum(axis=1))*3
word = word.merge(pre[['Prolific ID','dys-score']], on='Prolific ID', how='left' )

word['dys-score'] = word['dys-score'].fillna(-1)
word['dys-score'] = word['dys-score'].astype(int)

word = word.drop_duplicates(keep='first')

# add a column if they self-diagnosed themselves as dyslexic
word = word.merge(pre[['Have you ever been diagnosed with Dyslexia?','Prolific ID']], on='Prolific ID', how='left' )
word = word.rename(columns={'Have you ever been diagnosed with Dyslexia?':'self-diagnosed'})

# rename columns and merge it with the challanging word dataset
word_dataset = word_dataset.rename(columns={'word':'words'})
word = word.merge(word_dataset, on='words', how='left')

# extract the last difficulty of each word
word['last_difficulty'] = word['difficulty'].fillna('').astype(str).str.split(',').str[-1]

# extract ids from prolific demography datasets -> Project: dyslexia - Medical
medical = medical_dyslexic['Participant id'].to_list()
word['med_dys'] = word['Prolific ID'].isin(medical)


# extract ids from prolific demography datasets -> Project: Dyslexia
self_dyslex = self_dyslexic['Participant id'].to_list()
word['self_dys_prolific'] = word['Prolific ID'].isin(self_dyslex)

# remove unnecessary columns
word = word.drop(columns=['difficulty', 'timeStamps', 'reason', 'url', 'Server Timestamp'])

word = word[word['last_difficulty'].astype(str).str.strip() != '']
word['Prolific ID'][word['Prolific ID']=='66d9a9dfafa9f033be6ee60a\n']='66d9a9dfafa9f033be6ee60a'
pre['What is your gender?'][pre['What is your gender?'].isna()]='unknown'
pre['What is your age? (in years)'][pre['What is your gender?']=='unknown'] = 0


def safe_zipf(word):
    if isinstance(word, str):
        return zipf_frequency(word, 'en')
    else:
        return None
word['zipf'] = word['words'].apply(safe_zipf)

# Data Cleaning
pre['What is your gender?'] = pre['What is your gender?'].str.strip().str.lower()

gender_map = {
    'male': 'Male',
    'female': 'Female',
    'man': 'Male',
    'woman': 'Female'
    'woman': 'Female',
    'nonbinary': 'Non-binary',
    'non-binary': 'Non-binary',
    'transgender': 'Transgender',
    'm':'Male',
    'femal':'Female',
    'nonbinary transmasculine' : 'Non-binary',
    'f':'Female',
    'non binary' : 'Non-binary',
    'transmasculine' : 'Transgender'
}

pre['What is your gender?'] = pre['What is your gender?'].map(gender_map).fillna(pre['What is your gender?'])

pre['What is your gender?'].value_counts()


# Data Integration (pre-survey and word dataset)
print(pre['Prolific ID'].nunique(), word['Prolific ID'].nunique())

pre_list = pre['Prolific ID'].to_list()
word_list = word['Prolific ID'].to_list()

temp = word[~word['Prolific ID'].isin(pre_list)]
print(temp['Prolific ID'].nunique()) # must be 0

pre = pre[pre['Prolific ID'].isin(word_list)]
print(pre['Prolific ID'].nunique())


# Remove under affected participants

l = ['67ed8f12809e993b465ce944', '67ccc62a623f466a4ad19371']
word = word[~word['Prolific ID'].isin(l)]
print(word['Prolific ID'].nunique())
```

```
# Convert to datetime
word['date and time'] = pd.to_datetime(word['date and time'])

# Keep only participants with ≤ 55 ratings
counts = word.groupby('Prolific ID').size()
valid_ids = counts[counts <= 55].index
filtered = word[word['Prolific ID'].isin(valid_ids)]

# Identify participants who ONLY used "1" or "2"
only_1_2_ids = filtered.groupby('Prolific ID')['last_difficulty'].apply(
    lambda x: x.astype(str).str.strip().isin(['1','2']).all()
)
only_1_2_ids = only_1_2_ids[only_1_2_ids].index
print('len', len(only_1_2_ids))


# Compute durations for those participants
durations = filtered.groupby('Prolific ID')['date and time'].max() - filtered.groupby('Prolific ID')['date and time'].min()
durations_seconds = durations.dt.total_seconds()

careless_durations = durations_seconds.loc[only_1_2_ids]
```

```
# Compute cutoff = 25th percentile
quartiles = careless_durations.quantile([0.25, 0.5, 0.75])
cutoff = careless_durations.quantile(0.25)

print("Quartiles of their session durations (seconds):")
print(quartiles)
print("\nCutoff (25th percentile of careless durations):", cutoff, "seconds ≈", cutoff/60, "minutes")
# print(careless_durations.sort_values())


# Detect and remove the unreliable participants just selected 1
df = word.groupby('Prolific ID')['last_difficulty']
all_ones = df.apply(lambda s: (s == '1').all())
all_ones.value_counts()
num_participants_all_1 = int(all_ones.sum())
ids_all_1 = all_ones[all_ones].index.tolist()

print('only 1', num_participants_all_1)  # count and a peek at first 10 IDs

# word_all1 = word[word['Prolific ID'].isin(ids_all_1)].copy()

# Drop all rows from those participants
word = word[~word['Prolific ID'].isin(ids_all_1)].copy()

# Detect and remove the unreliable participants just selected 1 and 2 and did not take enough time
df1 = word.copy()
df1['last_difficulty'] = pd.to_numeric(df1['last_difficulty'], errors='coerce')
df1['date and time'] = pd.to_datetime(df1['date and time'], errors='coerce')
df1 = df1.dropna(subset=['last_difficulty', 'date and time'])

df = df1.groupby('Prolific ID')

only_12 = df['last_difficulty'].apply(lambda s: s.isin([1,2]).all())
dur_min = (df['date and time'].max() - df['date and time'].min()).dt.total_seconds() / 60
ids_3 = dur_min.index[only_12 & (dur_min < cutoff)].tolist()
print('ids_1,2',len(ids_3))


# Drop all rows from those participants
word = word[~word['Prolific ID'].isin(ids_3)].copy()
print(word['Prolific ID'].nunique())
mask = (
    (word['med_dys'] == True)|(word['dys-score'] >=12 ) )

word_dys = word[mask]
print('word_dys',word_dys['Prolific ID'].nunique())
word_notdys = word[~mask]
print('word_notdys',word_notdys['Prolific ID'].nunique())

word = pd.concat([word_dys, word_notdys], ignore_index=True)
print('word',word['Prolific ID'].nunique())


# Only for non-teachers

mask = (
    (word['med_dys'] == True)|(word['dys-score'] >=12 ) )
```

```
word_dys = word[mask]
print(word_dys['Prolific ID'].nunique())
word_notdys = word[~mask]
print(word_notdys['Prolific ID'].nunique())
dys_list= word_dys['Prolific ID'].to_list()
nondys_list = word_notdys['Prolific ID'].to_list()

word['dyslexia'] = False
word['dyslexia'] = word['Prolific ID'].isin(dys_list)

word = word.drop(columns=['self_dys_prolific','med_dys','self-diagnosed', 'familiarity', 'some letters', 'length'])
word['dyslexia'].value_counts(dropna = False)


pretemp = pre[['Prolific ID','What is your gender?','What is your age? (in years)']]
wordtemp = word[['Prolific ID','dyslexia']]
temp = wordtemp.merge(pretemp, on='Prolific ID', how='left')
temp = temp.dropna()
temp = temp.drop_duplicates(subset='Prolific ID')
temp['What is your age? (in years)'] = temp['What is your age? (in years)'].astype(int)
temp = temp[temp['What is your age? (in years)'] != 0]

temp.groupby(['dyslexia', 'What is your gender?'])['What is your age? (in years)'].describe()
```

```
# Normalized distribution to see what word feature rated as easy for dyslexic people

temp = word_dys.copy()


rating_counts = temp.groupby('words')['last_difficulty'].value_counts().unstack(fill_value=0)
rating_counts['total'] = rating_counts.sum(axis=1)

#add columns of 1 ,2,3,4 to a new dataset: rating_counts
rating_counts['1%'] = ((rating_counts['1'] / rating_counts['total']) * 100).round(2)
rating_counts['2%'] = ((rating_counts['2'] / rating_counts['total']) * 100).round(2)
rating_counts['3%'] = ((rating_counts['3'] / rating_counts['total']) * 100).round(2)
rating_counts['4%'] = ((rating_counts['4'] / rating_counts['total']) * 100).round(2)

rating_counts = rating_counts.merge(temp[['words','long_words', 'diagraphs', 'not_freq_words', 'silent_letters', 'vowel_digraphs','has_homophones', 'difficult_
rating_counts.drop_duplicates(subset='words', keep='first', inplace=True)
# print(rating_counts.shape)
# filter out the words that are rated as very easy with the higher rate than 80% and creating 2 new datasets: difficult_words, easy_words
difficult_words = rating_counts[
    ((rating_counts['total'] < 5) & (rating_counts['1']/ rating_counts['total']< 0.75)) |
    ((rating_counts['total'] >= 5) & (rating_counts['1'] / rating_counts['total'] < 0.8))
]
dys_difficult_list = difficult_words['words'].to_list()
# print(dys_difficult_list)

easy_words = rating_counts[
    ((rating_counts['total'] < 5) & (rating_counts['1']/ rating_counts['total']>= 0.75)) |
    ((rating_counts['total'] >= 5) & (rating_counts['1'] / rating_counts['total'] >= 0.8))
]

# print(rating_counts.shape,difficult_words.shape, easy_words.shape)

# Normalized distribution to see what word feature is the hardest(2,3,4) -  dyslexic people
```

```
categories = ['long_words', 'diagraphs', 'not_freq_words',
              'silent_letters', 'vowel_digraphs', 'has_homophones',
              'difficult_orthography']

normalized_stats = {}

for cat in categories:
    total_with_feature = rating_counts[rating_counts[cat] == 1].shape[0]
    difficult_with_feature = difficult_words[difficult_words[cat] == 1].shape[0]

    if total_with_feature > 0:
        normalized_stats[cat] = round(difficult_with_feature / total_with_feature * 100, 2)
    else:
        normalized_stats[cat] = 0

norm_dysdf = pd.DataFrame.from_dict(normalized_stats, orient='index', columns=['% Hard Ratings'])
norm_dysdf = norm_dysdf.sort_values(by='% Hard Ratings', ascending=False)

# Normalized distribution to see what word feature rated as easy for non-dyslexic people

temp = word_notdys.copy()

rating_counts = temp.groupby('words')['last_difficulty'].value_counts().unstack(fill_value=0)
rating_counts['total'] = rating_counts.sum(axis=1)

#add columns of 1 ,2,3,4 to a new dataset: rating_counts
rating_counts['1%'] = ((rating_counts['1'] / rating_counts['total']) * 100).round(2)
rating_counts['2%'] = ((rating_counts['2'] / rating_counts['total']) * 100).round(2)
rating_counts['3%'] = ((rating_counts['3'] / rating_counts['total']) * 100).round(2)
rating_counts['4%'] = ((rating_counts['4'] / rating_counts['total']) * 100).round(2)

rating_counts = rating_counts.merge(temp[['words','long_words', 'diagraphs', 'not_freq_words', 'silent_letters', 'vowel_digraphs','has_homophones', 'difficult_

rating_counts.drop_duplicates(subset='words', keep='first', inplace=True)


difficult_words = rating_counts[
    ((rating_counts['total'] < 5) & (rating_counts['1']/ rating_counts['total']< 0.75)) |
    ((rating_counts['total'] >= 5) & (rating_counts['1'] / rating_counts['total'] < 0.8))
]
nondys_difficult_list = difficult_words['words'].to_list()

easy_words = rating_counts[
    ((rating_counts['total'] < 5) & (rating_counts['1']/ rating_counts['total']>= 0.75)) |
    ((rating_counts['total'] >= 5) & (rating_counts['1'] / rating_counts['total'] >= 0.8))
]


# Normalized distribution to see what word feature is the hardest - non dyslexic people
categories = ['long_words', 'diagraphs', 'not_freq_words',
              'silent_letters', 'vowel_digraphs', 'has_homophones',
              'difficult_orthography']

normalized_stats = {}

for cat in categories:
    total_with_feature = rating_counts[rating_counts[cat] == 1].shape[0]
    difficult_with_feature = difficult_words[difficult_words[cat] == 1].shape[0]
    if total_with_feature > 0:
        normalized_stats[cat] = round(difficult_with_feature / total_with_feature * 100, 2)
    else:
        normalized_stats[cat] = 0

norm_nondysdf = pd.DataFrame.from_dict(normalized_stats, orient='index', columns=['% Hard Ratings'])
norm_nondysdf = norm_nondysdf.sort_values(by='% Hard Ratings', ascending=False)
```

```
# XGBOOST
word['dys hard'] = (word['words'].isin(dys_difficult_list)) & (word['dyslexia']==True).astype(int)
word['nondys hard'] = (word['words'].isin(nondys_difficult_list)) & (word['dyslexia']==False).astype(int)

dys_word = word[word['dyslexia']==True]
nondys_word = word[word['dyslexia']==False]

dys_word = dys_word.drop(columns=['nondys hard'])
nondys_word = nondys_word.drop(columns=['dys hard'])


dys_word['dys hard'] = dys_word['dys hard'].astype(int)
nondys_word['nondys hard'] = nondys_word['nondys hard'].astype(int)

dys_word1 = dys_word.drop(columns=['Prolific ID', 'words', 'letters', 'date and time', 'dys-score', 'last_difficulty','not_freq_words','dyslexia', 'diagraphs',
nondys_word1 = nondys_word.drop(columns=['Prolific ID', 'words', 'letters', 'date and time', 'dys-score', 'last_difficulty', 'not_freq_words','dyslexia', 'diag
dys_word1 = dys_word.drop(columns=['zipf'])
nondys_word1 = nondys_word.drop(columns=['zipf'])
dys_word1

# Dyslexic dataset
X_dys = dys_word1.drop(columns=['dys hard'])
y_dys = dys_word1['dys hard']

X_train_dys, X_test_dys, y_train_dys, y_test_dys = train_test_split(X_dys, y_dys, test_size=0.2, random_state=42)

model_dys = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss')
model_dys.fit(X_train_dys, y_train_dys)

y_pred_dys = model_dys.predict(X_test_dys)
print("Accuracy (Dyslexic):", accuracy_score(y_test_dys, y_pred_dys))

#  Non-dyslexic dataset
X_nondys = nondys_word1.drop(columns=['nondys hard'])
y_nondys = nondys_word1['nondys hard']

X_train_nondys, X_test_nondys, y_train_nondys, y_test_nondys = train_test_split(X_nondys, y_nondys, test_size=0.2, random_state=42)

model_nondys = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss')
model_nondys.fit(X_train_nondys, y_train_nondys)

y_pred_nondys = model_nondys.predict(X_test_nondys)
```

```
#CV - baseline
scoring = {
    "acc": "accuracy",
    "f1": "f1",
    "roc": "roc_auc",
    "pr": "average_precision",
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

clf = xgb.XGBClassifier(
    tree_method="hist",
    eval_metric="logloss",
    random_state=42
)

# Dyslexic
res_dys = cross_validate(clf, X_dys, y_dys, cv=cv, scoring=scoring, n_jobs=-1)
print("Dyslexic (5-fold):",
      f"Acc {np.mean(res_dys['test_acc']):.3f}",
      f"F1 {np.mean(res_dys['test_f1']):.3f}",
      f"ROC {np.mean(res_dys['test_roc']):.3f}",
      f"PR {np.mean(res_dys['test_pr']):.3f}")

# Non-dyslexic
res_nondys = cross_validate(clf, X_nondys, y_nondys, cv=cv, scoring=scoring, n_jobs=-1)
print("Non-dyslexic (5-fold):",
      f"Acc {np.mean(res_nondys['test_acc']):.3f}",
      f"F1 {np.mean(res_nondys['test_f1']):.3f}",
      f"ROC {np.mean(res_nondys['test_roc']):.3f}",
      f"PR {np.mean(res_nondys['test_pr']):.3f}")


def mean_std(arr):
    return f"{np.mean(arr):.3f} ± {np.std(arr):.3f}"

print("Dyslexic (5-fold):",
      f"Acc {mean_std(res_dys['test_acc'])}",
      f"F1 {mean_std(res_dys['test_f1'])}",
      f"ROC {mean_std(res_dys['test_roc'])}",
      f"PR {mean_std(res_dys['test_pr'])}")

print("Non-dyslexic (5-fold):",
      f"Acc {mean_std(res_nondys['test_acc'])}",
      f"F1 {mean_std(res_nondys['test_f1'])}",
      f"ROC {mean_std(res_nondys['test_roc'])}",
      f"PR {mean_std(res_nondys['test_pr'])}")
```

```
# XGBoost - seperate datasets

dys_word['dys hard'] = dys_word['dys hard'].astype(int)
nondys_word['nondys hard'] = nondys_word['nondys hard'].astype(int)



# remove some columns
dys_word = dys_word.drop(columns=['Prolific ID', 'words', 'letters', 'date and time', 'dys-score', 'last_difficulty','not_freq_words','dyslexia','reason explar
nondys_word = nondys_word.drop(columns=['Prolific ID', 'words', 'letters', 'date and time', 'dys-score', 'last_difficulty', 'not_freq_words','dyslexia','reasor
dys_word1 = dys_word.drop(columns=['Prolific ID', 'words', 'letters', 'date and time', 'dys-score', 'last_difficulty','not_freq_words','dyslexia', 'diagraphs',
nondys_word1 = nondys_word.drop(columns=['Prolific ID', 'words', 'letters', 'date and time', 'dys-score', 'last_difficulty', 'not_freq_words','dyslexia', 'diag
dys_word1 = dys_word.drop(columns=['zipf'])
nondys_word1 = nondys_word.drop(columns=['zipf'])
dys_word1

# Dyslexic dataset
X_dys = dys_word1.drop(columns=['dys hard'])
y_dys = dys_word1['dys hard']

X_train_dys, X_test_dys, y_train_dys, y_test_dys = train_test_split(X_dys, y_dys, test_size=0.2, random_state=42)

model_dys = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss')
model_dys.fit(X_train_dys, y_train_dys)

y_pred_dys = model_dys.predict(X_test_dys)
print("Accuracy (Dyslexic):", accuracy_score(y_test_dys, y_pred_dys))

#  Non-dyslexic dataset
X_nondys = nondys_word1.drop(columns=['nondys hard'])
y_nondys = nondys_word1['nondys hard']

X_train_nondys, X_test_nondys, y_train_nondys, y_test_nondys = train_test_split(X_nondys, y_nondys, test_size=0.2, random_state=42)

model_nondys = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss')
model_nondys.fit(X_train_nondys, y_train_nondys)

y_pred_nondys = model_nondys.predict(X_test_nondys)
print("Accuracy (Non-Dysleslexic):", accuracy_score(y_test_nondys, y_pred_nondys))
```

```
scoring = {
    "acc": "accuracy",
    "f1": "f1",
    "roc": "roc_auc",
    "pr": "average_precision",
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

clf = xgb.XGBClassifier(
    tree_method="hist",
    eval_metric="logloss",
    random_state=42
)

# Dyslexic
res_dys = cross_validate(clf, X_dys, y_dys, cv=cv, scoring=scoring, n_jobs=-1)
print("Dyslexic (5-fold):",
      f"Acc {np.mean(res_dys['test_acc']):.3f}",
      f"F1 {np.mean(res_dys['test_f1']):.3f}",
      f"ROC {np.mean(res_dys['test_roc']):.3f}",
      f"PR {np.mean(res_dys['test_pr']):.3f}")

# Non-dyslexic
res_nondys = cross_validate(clf, X_nondys, y_nondys, cv=cv, scoring=scoring, n_jobs=-1)
print("Non-dyslexic (5-fold):",
      f"Acc {np.mean(res_nondys['test_acc']):.3f}",
      f"F1 {np.mean(res_nondys['test_f1']):.3f}",
      f"ROC {np.mean(res_nondys['test_roc']):.3f}",
      f"PR {np.mean(res_nondys['test_pr']):.3f}")


def mean_std(arr):
    return f"{np.mean(arr):.3f} ± {np.std(arr):.3f}"

print("Dyslexic (5-fold):",
      f"Acc {mean_std(res_dys['test_acc'])}",
      f"F1 {mean_std(res_dys['test_f1'])}",
      f"ROC {mean_std(res_dys['test_roc'])}",
      f"PR {mean_std(res_dys['test_pr'])}")

print("Non-dyslexic (5-fold):",
      f"Acc {mean_std(res_nondys['test_acc'])}",
      f"F1 {mean_std(res_nondys['test_f1'])}",
      f"ROC {mean_std(res_nondys['test_roc'])}",
      f"PR {mean_std(res_nondys['test_pr'])}")
```

```
# Inferential
dys_ratings = word[word['dyslexia']==1]
nondys_ratings = word[word['dyslexia']==0]
nondys_ratings['Prolific ID'].nunique(), dys_ratings["Prolific ID"].nunique()

dys_ratings = word[word['dyslexia']==1]
nondys_ratings = word[word['dyslexia']==0]
nondys_ratings['Prolific ID'].nunique(), dys_ratings["Prolific ID"].nunique()


features = [
    'long_words', 'diagraphs', 'not_freq_words',
    'silent_letters', 'vowel_digraphs', 'has_homophones',
    'difficult_orthography'
]

df = word.copy()


df['last_difficulty_num'] = pd.to_numeric(df['last_difficulty'].astype(str).str.strip(), errors='coerce')
df = df[df['last_difficulty_num'].isin([1,2,3,4])]


df['binary'] = np.where(df['last_difficulty_num'] == 1, 'Very Easy', 'Not Very Easy')

df[features] = df[features].fillna(0).astype(int)

results = []

for feat in features:
    df_feat = df[df[feat] == 1].copy()

    table = pd.crosstab(df_feat['dyslexia'], df_feat['binary'])
    table = table.reindex(index=[True, False], columns=['Not Very Easy', 'Very Easy'], fill_value=0)
    # Fisher exact needs
    odds_ratio, p_value = fisher_exact(table.values)


    # Compare odds of Not Very Easy (dys vs non)

    a, b = table.loc[True,  'Not Very Easy'], table.loc[True,  'Very Easy']
    c, d = table.loc[False, 'Not Very Easy'], table.loc[False, 'Very Easy']

    dys_odds = (a / b) if b != 0 else np.inf
    non_odds = (c / d) if d != 0 else np.inf
    direction = "Dys higher Not Very Easy" if dys_odds > non_odds else "Non higher Not Very Easy"

    results.append({
        "feature": feat,
        "dys_NotVeryEasy": int(a),
        "dys_VeryEasy": int(b),
        "non_NotVeryEasy": int(c),
        "non_VeryEasy": int(d),
        "odds_ratio": odds_ratio,
        "p_raw": p_value,
        "direction": direction
    })

res = pd.DataFrame(results)

# FDR correction
res["p_fdr"] = multipletests(res["p_raw"], method="fdr_bh")[1]
res["significant_fdr_0.05"] = res["p_fdr"] < 0.05

# sort by p
res = res.sort_values("p_fdr")
res
```

```
features = [
    'long_words', 'diagraphs', 'not_freq_words',
    'silent_letters', 'vowel_digraphs', 'has_homophones',
    'difficult_orthography'
]

df = word.copy()

df['last_difficulty_num'] = pd.to_numeric(df['last_difficulty'].astype(str).str.strip(), errors='coerce')
df = df[df['last_difficulty_num'].isin([1,2,3,4])]

df['binary'] = np.where(df['last_difficulty_num'] == 1, 'Very Easy', 'Not Very Easy')
df[features] = df[features].fillna(0).astype(int)

results = []

Z = 1.96

for feat in features:
    df_feat = df[df[feat] == 1].copy()
```

```
        table = pd.crosstab(df_feat['dyslexia'], df_feat['binary'])
        table = table.reindex(index=[True, False], columns=['Not Very Easy', 'Very Easy'], fill_value=0)

        # counts
        a = int(table.loc[True,  'Not Very Easy'])
        b = int(table.loc[True,  'Very Easy'])
        c = int(table.loc[False, 'Not Very Easy'])
        d = int(table.loc[False, 'Very Easy'])

        # fisher exact p-value + OR
        odds_ratio, p_value = fisher_exact([[a, b], [c, d]])

        if b == 0 or d == 0:
            dys_odds = (a + 0.5) / (b + 0.5)
            non_odds = (c + 0.5) / (d + 0.5)
        else:
            dys_odds = a / b
            non_odds = c / d
        direction = "Dys higher Not Very Easy" if dys_odds > non_odds else "Non higher Not Very Easy"


        if (a == 0) or (b == 0) or (c == 0) or (d == 0):
            a_ci, b_ci, c_ci, d_ci = a + 0.5, b + 0.5, c + 0.5, d + 0.5
        else:
            a_ci, b_ci, c_ci, d_ci = float(a), float(b), float(c), float(d)

        # log(OR) and SE
        or_ci = (a_ci * d_ci) / (b_ci * c_ci)
        se = np.sqrt(1/a_ci + 1/b_ci + 1/c_ci + 1/d_ci)
        log_or = np.log(or_ci)

        ci_low = np.exp(log_or - Z * se)
        ci_high = np.exp(log_or + Z * se)

        results.append({
            "feature": feat,
            "dys_NotVeryEasy": a,
            "dys_VeryEasy": b,
            "non_NotVeryEasy": c,
            "non_VeryEasy": d,
            "odds_ratio": odds_ratio,
            "ci95_low": ci_low,
            "ci95_high": ci_high,
            "p_raw": p_value,
            "direction": direction
        })

    res = pd.DataFrame(results)

    # fdr correction
    res["p_fdr"] = multipletests(res["p_raw"], method="fdr_bh")[1]
    res["significant_fdr_0.05"] = res["p_fdr"] < 0.05

    res = res.sort_values("p_fdr")

    res
```

```
res_fmt = res.copy()

res_fmt["OR"] = res_fmt["odds_ratio"].map(lambda x: f"{x:.2f}")
res_fmt["95% CI"] = res_fmt.apply(lambda r: f"[{r['ci95_low']:.2f}, {r['ci95_high']:.2f}]", axis=1)

def fmt_p(p):
    return "< .001" if p < 1e-3 else f"{p:.3f}"
res_fmt["p (FDR)"] = res_fmt["p_fdr"].map(fmt_p)

res_fmt["Dys (NVE/VE)"] = res_fmt.apply(lambda r: f"{int(r['dys_NotVeryEasy'])}/{int(r['dys_VeryEasy'])}", axis=1)
res_fmt["Non (NVE/VE)"] = res_fmt.apply(lambda r: f"{int(r['non_NotVeryEasy'])}/{int(r['non_VeryEasy'])}", axis=1)

final_table = res_fmt[["feature", "Dys (NVE/VE)", "Non (NVE/VE)", "OR", "95% CI", "p (FDR)"]]
final_table
```

```
# teachers data

teachers_words = gc.open_by_url('https://docs.google.com/spreadsheets/d/1BgHVgMfZjQkiA8rJLfuPlXGCJ6S5w138IlO-0kCS0SA/edit?gid=0#gid=0')
teachers_pre = gc.open_by_url('https://docs.google.com/spreadsheets/d/1XoA_8w8mI5LdhD8sIahDba2_OhdpGZZQvmLzl7sq1Cw/edit?gid=1328307878#gid=1328307878')
teachers_post = gc.open_by_url('https://docs.google.com/spreadsheets/d/1o3Tz4DAnyh7__GsXxM0Ir0zvFR60rjicwibX10-kwxE/edit?gid=575842559#gid=575842559')

wordsheet = teachers_words.worksheet('Sheet1')
presheet = teachers_pre.worksheet('Sheet1')
postsheet = teachers_post.worksheet('Sheet1')

tword = get_as_dataframe(wordsheet)
tpre = get_as_dataframe(presheet)
tpost = get_as_dataframe(postsheet)

tword = tword.drop(columns={'Unnamed: 10','Unnamed: 11'})
tword.shape, tpre.shape, tpost.shape, tword.columns

# Teacher - pre processing
def safe_zipf(word):
    if isinstance(word, str):
        return zipf_frequency(word, 'en')
    else:
        return None

tpre = tpre.rename(columns={'Please enter your Prolific ID:':'Prolific ID'})
tpre = tpre.drop_duplicates(subset='Prolific ID')
duplicate_ids = tpre['Prolific ID'][tpre['Prolific ID'].duplicated()]
print(tword.columns)
tpre = tpre[tpre['grade category'].isna()== False]

tword = tword.rename(columns={'word':'words'})

tword['last_difficulty'] = tword['difficulty'].fillna('').astype(str).str.split(',').str[-1]

tword['zipf'] = tword['words'].apply(safe_zipf)

tword = tword.drop(columns=['difficulty', 'timeStamps', 'reason', 'url', 'Server Timestamp'])
tword.columns
```

```
# teachers - data consistency
tpre['What is your gender?'] = tpre['What is your gender?'].str.strip().str.lower()
```

```
gender_map = {
    'nb': 'unknown',
    'chick': 'unknown',
    'male': 'Male',
    'female': 'Female',
    'man': 'Male',
    'woman': 'Female',
    'nonbinary': 'Non-binary',
    'non-binary': 'Non-binary',
    'transgender': 'Transgender',
    'm':'Male',
    'femal':'Female',
    'female/ woman' : 'Female',
    'F':'Female',
    'femaile':'Female',
    'fEMALE':'Female',
    'Email' : 'Female',
    'f': 'Female','email': 'Female'
}

tpre['What is your gender?'] = tpre['What is your gender?'].map(gender_map).fillna(tpre['What is your gender?'])

tpre['What is your gender?'].value_counts()

# Teachers - data integration (pre-survey and word dataset)
print(tpre['Prolific ID'].nunique(), tword['Prolific ID'].nunique())

tpre_list = tpre['Prolific ID'].to_list()
tword_list = tword['Prolific ID'].to_list()

temp = tword[~tword['Prolific ID'].isin(tpre_list)]
print(temp['Prolific ID'].nunique()) # must be 0

tpre = tpre[tpre['Prolific ID'].isin(tword_list)]
print(tpre['Prolific ID'].nunique())

#mean, min, max
tpre['What is your age? (in years)'] = tpre['What is your age? (in years)'].astype(int)
tpre = tpre[tpre['What is your age? (in years)'] != 0]

tpre1 = tpre[tpre['What is your gender?'].isin(['Male', 'Female'])]
tpre1['What is your age? (in years)'].astype(int).describe()
```

```
# Teachers- Quartile-Based Adjustment:

tword['date and time'] = pd.to_datetime(tword['date and time'])

counts = tword.groupby('Prolific ID').size()
valid_ids = counts[counts <= 55].index
filtered = tword[tword['Prolific ID'].isin(valid_ids)]

only_1_2_ids = filtered.groupby('Prolific ID')['last_difficulty'].apply(
    lambda x: x.astype(str).str.strip().isin(['1','2']).all()
)
only_1_2_ids = only_1_2_ids[only_1_2_ids].index
print('len', len(only_1_2_ids))
```

```python
durations = filtered.groupby('Prolific ID')['date and time'].max() - filtered.groupby('Prolific ID')['date and time'].min()
durations_seconds = durations.dt.total_seconds()

careless_durations = durations_seconds.loc[only_1_2_ids]

# Compute cutoff = 25th percentile
quartiles = careless_durations.quantile([0.25, 0.5, 0.75])
cutoff = careless_durations.quantile(0.25)

print("Quartiles of their session durations (seconds):")
print(quartiles)
print("\nCutoff (25th percentile of careless durations):", cutoff, "seconds ≈", cutoff/60, "minutes")
print(careless_durations.sort_values())

df = tword.groupby('Prolific ID')['last_difficulty']
all_ones = df.apply(lambda s: (s == '1').all())
all_ones.value_counts()
num_participants_all_1 = int(all_ones.sum())
ids_all_1 = all_ones[all_ones].index.tolist()


tword = tword[~tword['Prolific ID'].isin(ids_all_1)].copy()

# remove the unreliable participants just selected 1 and 2 and did not take enough time
df1 = tword.copy()
df1['last_difficulty'] = pd.to_numeric(df1['last_difficulty'], errors='coerce')
df1['date and time'] = pd.to_datetime(df1['date and time'], errors='coerce')
df1 = df1.dropna(subset=['last_difficulty', 'date and time'])

df = df1.groupby('Prolific ID')

only_12 = df['last_difficulty'].apply(lambda s: s.isin([1,2]).all())
dur_min = (df['date and time'].max() - df['date and time'].min()).dt.total_seconds() / 60
ids_3 = dur_min.index[only_12 & (dur_min < cutoff)].tolist()


print('len', tword['Prolific ID'].nunique() )

tword = tword[~tword['Prolific ID'].isin(ids_3)].copy()
print(tword['Prolific ID'].nunique())

# teacher's - XGBoost

# tword['hard'] = [0 if tword['last_difficulty'][i] == '1' else 1 for i in range(len(tword))]
# tword= tword.drop(columns=['Prolific ID', 'words', 'letters', 'reason explanation', 'date and time', 'last_difficulty' ])
X_teacher = tword.drop(columns=['hard'])
y_teacher = tword['hard']


scoring = {
    "acc": "accuracy",
    "f1": "f1",
    "roc": "roc_auc",
    "pr": "average_precision",
}

neg, pos = np.bincount(y_teacher)
scale_pos_weight = neg / pos
scale_pos_weight = neg / pos

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

clf = xgb.XGBClassifier(
    tree_method="hist",
    eval_metric="logloss",
    random_state=42,
    n_estimators=500,
    learning_rate=0.05,
    max_depth=4,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_lambda=1.0,
    scale_pos_weight=scale_pos_weight,
    n_jobs=-1
)






teacher_res = cross_validate(clf, X_teacher, y_teacher, cv=cv, scoring=scoring, n_jobs=-1)
print("teachers (5-fold):",
      f"Acc {np.mean(teacher_res['test_acc']):.3f}",
      f"F1 {np.mean(teacher_res['test_f1']):.3f}",
      f"ROC {np.mean(teacher_res['test_roc']):.3f}",
      f"PR {np.mean(teacher_res['test_pr']):.3f}")


def mean_std(arr):
    return f"{np.mean(arr):.3f} ± {np.std(arr):.3f}"

print("Teachers (5-fold):",
      f"Acc {mean_std(teacher_res['test_acc'])}",
      f"F1 {mean_std(teacher_res['test_f1'])}",
      f"ROC {mean_std(teacher_res['test_roc'])}",
      f"PR {mean_std(teacher_res['test_pr'])}")
```