EDITAL DE AUXÍLIO À PESQUISA A RECÉM-DOUTORAS(ES) – 2022
ASSOCIAÇÃO BRASILEIRA DE CIÊNCIA POLÍTICA

RELATÓRIO CIENTÍFICO DA PESQUISA

Virginia Rocha da Silva

2023

1 Introdução

O presente relatório tem como objetivo atender a um dos requisitos de prestação de contas com relação ao valor de R$ 3.500,00 (Três Mil e Quinhentos Reais), concedido pelo Edital de Auxílio à Pesquisa a Recém-Doutoras(es), em 2022, do qual a presente autora foi uma das contempladas. O projeto de pesquisa aprovado para o recebimento do referido auxílio definiu como esperados os seguintes produtos acadêmicos: i) banco de dados sobre codificação de prefeitos dinásticos; ii) artigo científico sobre mensuração de dinastias políticas; iii) artigo científico sobre a relação entre dinastias e qualidade do governo; iv) transparência e replicabilidade. Neste relatório detalharei o desenvolvimento da pesquisa de acordo com estes objetivos, considerado a execução do que foi planejado e as modificações necessárias em relação ao que havia sido planejado. A sessão a seguir discute essas etapas da pesquisa e o que já foi produzido. Dois apêndices apresentados ao final do relatório apresentam i) *script* de coleta dos dados via *Google Custom Search JSON API* e ii) *script* de raspagem do texto coletado e pré-processamento do texto (com função de identificação dos parentescos entre políticos brasileiros nos municípios brasileiros).

2 Detalhamento do desenvolvimento da pesquisa

É importante ressaltar que o cronograma sugerido no projeto aprovado para recebimento do auxílio de pesquisa se baseou no nível de desenvolvimento da pesquisa. Neste sentido, um fator crucial para a finalização dos artigos de mensuração de dinastias e da relação entre dinastias e qualidade do governo se referia à coleta de dados adicionais sobre dinastias políticas. Não obstante, a pesquisadora verificou alguns desafios com relação à coleta. O primeiro deles se refere ao aprimoramento dos scripts de coleta de dados via API do *Google Search*. A expectativa seria utilizar o script desenvolvido pela pesquisadora em sua tese de doutorado, mas a função de coleta de dos dados apresentou uma série de erros relacionados à intensidade de *queries* pelo API. Neste sentido, foi necessário adaptar o script original de forma a inserir *time.sleep* sistemáticos que permitiriam prevenir a interrupção da coleta de dados por conta de erros levantados pela API. O *script* corrigido e atualizado está disponível no APÊNDICE A. O banco de dados resultante desta coleta pode ser acessado neste link e abrange os dois candidatos mais votados nas eleições municipais de 2012 e 2016. A expansão do recorte de dados para além de candidatos eleitos permite o uso de estratégias de inferência causal mais rigorosas, como a regressão descontínua.

O segundo desafio enfrentado na codificação de políticos como dinásticos ou não diz respeito ao processo posterior a coleta de dados via API do *Google*: a raspagem de dados dos links gerados pela API do google, visto que a referida aplicação retorna: *tag* de pesquisa feita no google (nome completo do candidato + cidade + estado), link do resultado de busca no *Google*, título do link e *snippet* do link (excerto do texto). Embora o título e o *snippet* já traga resultados relevantes, o conteúdo completo do link agrega informações relevantes para a identificação de menção a parentesco entre um determinando candidato e um político. O *script* atualizado, disponível no APÊNDICE B, já permite a extração do conteúdo dos links fornecidos pela API do *Google*, mas a raspagem acontece de forma muito lenta e extrai todas as informações do link (incluindo ícones de menu, cabeçalho etc.). A pesquisadora e o assistente de pesquisa contratado para apoiar a codificação dos políticos como dinásticos estão verificando duas possibilidades: i) refinar o código a fim de torná-lo mais rápido; ii) explorar a possibilidade de utilizar modelos generativos, como o *Chat-GPT4*[1], para a execução da extração de textos dos links já coletados.

Visto que a etapa de codificação dos candidatos não ocorreu no período planejado (primeiro semestre de 2023), não foi possível finalizar o artigo empírico sobre a relação entre dinastias políticas e qualidade do governo. Não obstante, a pesquisadora está em fase de finalização do capítulo provisoriamente intitulado "Famílias na Política: Uma Revisão sobre a Definição, Mensuração e Efeitos de Dinastias Políticas no Brasil e no Mundo" como parte do livro "Instituições, Políticas e Governo", organizado pela Profa. Mariana Batista e os doutorandos do seu grupo de pesquisa (homônimo ao livro), Pedro Nascimento e Bhreno Vieira. Este capítulo, cuja publicação está prevista para 2024 pela Editora da Universidade Estadual da Paraíba (EDUEPB), versa sobre a produção acadêmica nas ciências sociais, com ênfase na ciência política, sobre dinastias políticas. A pesquisadora utilizada o *Google Scholar* e a plataforma *Consensus* para identificar trabalhos que exploram os efeitos de dinastias políticas sobre a qualidade no governo. Este último conceito é amplamente abordado, abrangendo desempenho econômico e irregularidades no governo, por exemplo. Os estudos foram sistematizados de acordo com a pergunta e o desenho de pesquisa, a forma pela qual as autoras/es definem dinastias (quando isso é explicitamente apresentado), pela maneira como autoras/es

[1] É importante ressaltar que o uso do *Chat-GPT4*, se viável, não fere preceitos éticos de pesquisa já que lida com dados publicamente disponíveis no *Google* (sem violação de dados sensíveis nem da Lei Geral de Proteção de Dados), bem como não configura a produção de material intelectual (como análise de resultados ou produção de argumento teórico). O uso dessa inteligência artificial teria o único objetivo de automatizar a atividade de extração de texto dos links do *Google* de forma potencialmente mais efetiva que o *Beautiful Soup* (biblioteca do Python atualmente utilizada no *script* de raspagem de texto, conforme Apêndice B).

identificam os políticos dinásticos (registrando estratégias e bancos de dados usados) e pelos resultados encontrados pelos estudos, apontando, quando possível, se os achados indicam efeitos positivos, negativos ou neutros sobre a qualidade do governo (compreendida de forma ampla).

Esse capítulo traz uma contribuição relevante para a literatura sobre dinastias em dois sentidos: i) ao sistematizar em Língua Portuguesa uma literatura que é, em seu *mainstream*, majoritariamente escrita na Língua Inglesa, ii) ao sistematizar a literatura de dinastias políticas incluindo análise sobre Brasil e identificando e sugerindo possíveis formas de políticos dinásticos no país. É importante ressaltar que a autora inseriu uma nota de agradecimento ao apoio recebido no desenvolvimento de sua pesquisa sobre dinastias no Brasil, o que inclui o auxílio concedido pela ABCP para o presente projeto de pesquisa.

Neste sentido, os próximos passos deste projeto, planejado para ser conclusão em 2024, incluem a finalização da codificação dos candidatos (na qual o assistente de pesquisa contratado está atuando para a construção de uma amostra de dados anotados para treinamento dos modelos de classificação supervisionada, neste momento com base nos *snippet* e, posteriormente, a partir do conteúdo raspado dos links do *Google*) das eleições municipais de 2012 e 2016 como dinásticos para finalização do banco de dados da pesquisa, que inclui a variável dependente de qualidade de governo mensurada via irregularidades identificadas nos relatórios da CGU e realização de análise do efeito de prefeitos dinásticos sobre a qualidade do governo, com a realização de regressão descontínua como estratégia de inferência causal. Tendo em vista que o capítulo teórico produzido sobre mensuração e efeito de dinastias dialogará diretamente com a literatura brasileira, a pesquisadora almeja submeter o artigo empírico da relação entre dinastias e qualidade do governo para uma revista internacional. É crucial ressaltar também que a autora possui uma agenda de pesquisa mais ampla com foco nas dinastias políticas e que qualquer trabalho produzido com base nos dados coletados com auxílio do financiamento da ABCP incluirá os devidos agradecimentos ao apoio financeiro concedido pela instituição.

APÊNDICE A – FUNÇÃO APRIMORADA PARA NOVA COLETA DE DADOS VIA API DO *GOOGLE*

```
import numpy as np
# Create list from df's variables
df1['search_tag'] = df1["name"] + ' ' + df1["city"] + ' ' + df1["state"]
df1['search_tag']

# Other possible search tags
# 'Família'  + ' ' +  df["name"] + ' ' + df["city"] + ' ' + df["state"]
# 'Herdeiro Político'  + ' ' +  df["name"] + ' ' + df["city"] + ' ' +
df["state"]
```

```
# Call key
devKey = <incluir dev key>
```

```
def searchCandidate(search_tag_list, candidate_tag_list, dev_key):
    ''' Function to execute search and return dataset with candidate,
search_tag, title, snippet, link.

        search_tag_list: list with search tags
        candidate_tag_list: list with candidate names
        obs: search_tag_list and candidate_tag_list must have the same
length
    '''
    # Check if lengths of the lists match
    if len(search_tag_list) != len(candidate_tag_list):
        raise ValueError("search_tag_list and candidate_tag_list must have
the same length")

    # Define parameters to build service
    customSearchEngineId = '6fc9a792fba55225d'
    country = 'countryBR'  # restricts search for Brazil
    language = 'lang_pt'  # return results in Portuguese only
    termsToExclude = ['médicos jus brasil google maps']
    numberOfResults = 10
    service = build("customsearch", "v1", developerKey=dev_key)

    # Initialize search dictionary
    search = {'candidate': [], 'search_tag': [], 'items': []}
```

```
    # Step 1 - get results from page one
    for i in range(len(search_tag_list)):
        try:
            p1 = service.cse().list(q=search_tag_list[i],
cx=customSearchEngineId, num=numberOfResults, excludeTerms=termsToExclude,
cr=country, lr=language).execute()
            search['items'].append(p1.get('items', 'NOT FOUND'))
        except Exception as e:
            print(f"Error fetching page 1 for {search_tag_list[i]}: {e}")
            search['items'].append('NOT FOUND')
        search['candidate'].append(candidate_tag_list[i])
        search['search_tag'].append(search_tag_list[i])

    # Step 2 - get results from page two (by including in p2 the argument
'start=11')
    for i in range(len(search_tag_list)):
        try:
            p2 = service.cse().list(q=search_tag_list[i], start=11,
cx=customSearchEngineId, num=numberOfResults, excludeTerms=termsToExclude,
cr=country, lr=language).execute()
            search['items'].append(p2.get('items', 'NOT FOUND'))
        except Exception as e:
            print(f"Error fetching page 2 for {search_tag_list[i]}: {e}")
            search['items'].append('NOT FOUND')
        search['candidate'].append(candidate_tag_list[i])
        search['search_tag'].append(search_tag_list[i])

    # Check if lengths of lists in search are consistent
    print("Lengths:", len(search['candidate']), len(search['search_tag']),
len(search['items']))

    # Step 3 get title, snippet, and link from each item
    export_data = {'candidate': [], 'search_tag': [], 'title': [],
'snippet': [], 'link': []}
    for i in range(len(search['items'])):
        items = search['items'][i]

        if items == 'NOT FOUND':
            export_data['candidate'].append(search['candidate'][i])
            export_data['search_tag'].append(search['search_tag'][i])
            export_data['title'].append('NOT FOUND')
            export_data['snippet'].append('NOT FOUND')
            export_data['link'].append('NOT FOUND')
        else:
```

```
            for item in items:
                export_data['candidate'].append(search['candidate'][i])
                export_data['search_tag'].append(search['search_tag'][i])
                export_data['title'].append(item.get('title', 'NOT
FOUND'))
                export_data['snippet'].append(item.get('snippet', 'NOT
FOUND'))
                export_data['link'].append(item.get('link', 'NOT FOUND'))

    # This return statement must be indented to align with the function's
scope
    return pd.DataFrame(export_data)
```

APÊNDICE B – SCRIPT APRIMORADO PARA RASPAGEM DE TEXTO

```
# Import libraries

import bs4 #BeautifulSoup
import requests #To request data with bs4
import re #Regular expressions
import pandas as pd #To import and manipulate datasets
import numpy as np #To create lists
import time #For time.sleep
import nltk #For tokenization, parsing, classification, stemming, tagging and semantic reasoning
nltk.download('stopwords')
import os



def cleanText(text, tokenize=False):
    '''
    Standardize text to extract words.

    Args:
    text (str): Text to be cleaned.
    tokenize (bool): If True, returns a list of words; otherwise, returns a concatenated string.

    Returns:
    str or list: Cleaned text, either as a string or list of tokens.
    '''
    # Check if input is a string
    if not isinstance(text, str):
        raise ValueError("Input must be a string.")

    try:
        # Text to lowercase
```

```
        text = text.lower()

        # Remove numbers
        text = ''.join([i for i in text if not i.isdigit()])

        # Remove punctuation
        from nltk.tokenize import RegexpTokenizer
        tokenizer = RegexpTokenizer(r'\w+')  # Preserve words and alphanumeric
        text = tokenizer.tokenize(text)

        # Remove stopwords
        from nltk.corpus import stopwords
        stop = set(stopwords.words('portuguese'))
        text = [w for w in text if not w in stop]

        # Concatenate text if tokenization is false
        if not tokenize:
            text = ' '.join(text)

        return text

    except Exception as e:
        # Handle general errors in processing
        raise RuntimeError(f"Error occurred in text cleaning: {e}")


## Função getTextFromWebpageV2 para extrair texto de link do GoogleAPI

from google.colab import drive
drive.mount('/content/drive')
```

```
df =
pd.read_csv('/content/drive/MyDrive/Paper_Dyn_Patronage/combined_dataset_10nov23_google
api_collection.csv')

df.head(5)

def getTextFromWebpageV2(link, candidate, search_tag):
    '''
    Process a single link to extract text, and return the data as a dictionary.

    Args:
    link (str): The webpage link.
    candidate (str): The candidate name.
    search_tag (str): The search tag.

    Returns:
    dict: Dictionary containing the processed data for a single link.
    '''
    result = {'candidate': candidate, 'search_tag': search_tag, 'page_link': link,
              'page_text_raw': '', 'page_text_clean': '', 'page_text_clean_token': ''}

    if not link or pd.isna(link):
        print(f"Skipping empty or invalid link: {link}")
        return result

    try:
        print(f"Processing link: {link}")
        # Skip LinkedIn links
        if "linkedin.com" in link:
            print(f"Skipping LinkedIn link (requires login): {link}")
            return result
```

```
            req_deep = requests.get(link, timeout=10)
            req_deep.raise_for_status()
            page_deep = bs4.BeautifulSoup(req_deep.text, "html.parser")
            text = page_deep.get_text()
            result['page_text_raw'] = text
            result['page_text_clean'] = cleanText(text)
            result['page_text_clean_token'] = cleanText(text, tokenize=True)
        except Exception as e:
            print(f"Exception for link {link}: {e}")
            result['page_text_raw'] = 'ERROR'
            result['page_text_clean'] = 'ERROR'
            result['page_text_clean_token'] = 'ERROR'

        return result

# Define directory
output_dir_path = '/content/drive/MyDrive/Paper_Dyn_Patronage'

def process_data(df, output_dir_path, file_name='texto_from_api.csv', increment=100):
    drive.mount('/content/drive')
    if not os.path.exists(output_dir_path):
        os.makedirs(output_dir_path)

    all_data = []
    for start in range(0, len(df), increment):
        end = min(start + increment, len(df))
        print(f"Processing range: {start}-{end}")

        for index, row in df.iloc[start:end].iterrows():
            link_data = getTextFromWebpageV2(row['link'], row['candidate'], row['search_tag'])
            all_data.append(link_data)
```

```
        # Save the accumulated data for the current partition
        partition_df = pd.DataFrame(all_data)
        partition_file_path = os.path.join(output_dir_path, f'{file_name[:-4]}_{end}{file_name[-4:]}')
        partition_df.to_csv(partition_file_path, index=False, escapechar='\\')
        print(f"Data saved to {partition_file_path}")

# Example usage
# df = ...  # Load your DataFrame here
process_data(df, '/content/drive/MyDrive/Paper_Dyn_Patronage')



# teste em slide do dataset
#part_df = df[0:5]
#data = getTextFromWebpageV2(part_df["candidate"], part_df['search_tag'], part_df["link"])
#data.head()

data.to_csv("data/teste.csv")

## Concatenate data - Coding test (performed by Bhreno) with 1000 texts and apply kinship function

# Mount drive to import data from drive folder
from google.colab import drive
drive.mount('/content/drive')

# Import datasets
# Import dfs
import pandas as pd
df = pd.read_csv('/content/drive/MyDrive/Paper_Dyn_Patronage/texto_from_api_1000.csv')

df.info()
```

```
# Convert columns it from float to  string type.
df['page_text_raw'] = df['page_text_raw'].apply(str)
df['page_text_clean'] = df['page_text_clean'].apply(str)
df['search_tag'] = df['search_tag'].apply(str)

# Clean text

!pip install pandas unidecode

import pandas as pd
import re
from unidecode import unidecode

def clean_text(text):
    # Convert text to lowercase
    text = text.lower()

    # Remove accents
    text = unidecode(text)

    # Remove digits and special characters (keeping only alphabetic characters and spaces)
    text = re.sub(r'[^a-z\s]', '', text)

    return text

# Apply the function to the 'page_text_raw' column
df['page_text_raw'] = df['page_text_raw'].astype(str).apply(clean_text)

# Display the cleaned text
print(df['page_text_raw'].head())
```

```
# Maximize column width so it show more of the page text
pd.options.display.max_colwidth = 100

df.head(30)

# Kinship function - updated with new terms gathered from test with Chat-GPT4

# Tested coding of text snippets with Chat-GPT4 for João Campos (Mayor in Recife, Brazil):
# 1. Asked the AI to collect 10 first google results about João Campos and return google results that indicated kinship based on the terms of the kinship function
# 2. Then, asked chat-GPT4 to display excerpts from google results that indicated kinship
# 3. Finally, asked chat-GPT4 to list terms that indicate dynastic kinship in politics. The result was the following:

#"filho de" or "filha de" (son of/daughter of)
#"neto de" or "neta de" (grandson of/granddaughter of)
#"pai" or "mãe" (father/mother)
#"tio" or "tia" (uncle/aunt)
#"primo" or "prima" (cousin)
#"sobrinho" or "sobrinha" (nephew/niece)
#"esposo" or "esposa" (husband/wife)
#"marido" (husband)
#"casado com" or "casada com" (married to)
#"herdeiro político" (political heir)
#"dinastia política" (political dynasty)
#"família política" (political family)
#"legado político" (political legacy)
#"tradição política" (political tradition)
#"política no sangue" (politics in the blood)
#"clã político" (political clan)
#"história na política" (history in politics)
#"laços políticos" (political ties)
```

```
#"influência política" (political influence)

# Loop to identify politicans who are son/daughter of ex-politicians

def identifyDynasty(text, name):
    if ' filho ' in text and ' filho ' not in name:
        new_text = 'son'
    elif ' filho do ' in text or ' filho de ' in text or ' filho da ' in text:
        new_text = 'son'
    elif ' filha do ' in text or ' filha de ' in text or ' filha da ' in text:
        new_text = 'daughter'
    elif ' filha ' in text:
        new_text = 'daughter'
    elif ' neto de ' in text or ' neta de ' in text:
        new_text = 'grandchild'
    elif ' pai ' in text or ' mãe ' in text:
        new_text = 'parent'
    elif ' tio ' in text or ' tia ' in text:
        new_text = 'uncle/aunt'
    elif ' primo ' in text or ' prima ' in text:
        new_text = 'cousin'
    elif ' sobrinho ' in text or ' sobrinha ' in text:
        new_text = 'nephew/niece'
    elif ' esposo ' in text or ' esposa ' in text or ' marido ' in text:
        new_text = 'spouse'
    elif ' casado com ' in text or ' casada com ' in text:
        new_text = 'married'
    elif ' herdeiro político ' in text:
        new_text = 'political heir'
    elif ' dinastia política ' in text:
        new_text = 'political dynasty'
    elif ' família política ' in text:
```

```
        new_text = 'political family'
    elif ' legado político ' in text:
        new_text = 'political legacy'
    elif ' tradição política ' in text:
        new_text = 'political tradition'
    elif ' política no sangue ' in text:
        new_text = 'politics in the blood'
    elif ' clã político ' in text:
        new_text = 'political clan'
    elif ' história na política ' in text:
        new_text = 'history in politics'
    elif ' laços políticos ' in text:
        new_text = 'political ties'
    elif ' influência política ' in text:
        new_text = 'political influence'
    else:
        new_text = 'non-dynastic'

    return new_text



# Apply function
df['dynasty_result'] = df.apply(lambda row: identifyDynasty(row['page_text_raw'], 'candidate'),
axis=1)

df.head(50)

# Chekck if any text indicates kinship
category_counts_sorted = df['dynasty_result'].value_counts().sort_index()

print(category_counts_sorted)
```

```
print(df.columns)

import pandas as pd

# Filter out rows where 'dynasty_result' is not 'non_dynastic'
filtered_df = df[df['dynasty_result'] != 'non_dynastic']

# Define the file path in your Google Drive
file_path = '/content/drive/MyDrive/Paper_Dyn_Patronage/test_dynastickinshipresult.csv'

# Save the filtered DataFrame to the CSV file in Google Drive
filtered_df.to_csv(file_path, index=False) # Set index=False if you don't want to save the index.



#from google.colab import files
#files.download('test_dynastickinshipresult.csv')
```