Paper Replication: "Seeing beyond the Trees: Using Machine Learning to Estimate the Impact of Minimum Wages on Labor Market Outcomes" (Cengiz, D., Dube, A., Lindner, A. and Zentler-Munro, D., 2022)

Lucia Gomez Llactahuamani
(University of Bonn)

This project replicates the first half of the main results from Cengiz, D., Dube, A., Lindner, A., & Zentler-Munro, D. (2022). Following the authors, I apply machine learning methods to identify the potencial workers who are actually affected by the minimum wage policy. Although the code for the second part of the paper – estimating the impact of the minimum wage on labor market outcomes – is still being developed, this project represents a significant advance. In contrast to the original code, which was written in Stata and R, I replicated the study in Python, streamlining all the code into a single programming language that is both free and open source.

[1]:
```
!pip install -q statsmodels
!pip install -q scikit-learn
!pip install -q plotly
```

[2]:
```
# Load packages
import pandas as pd
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf
import plotly.graph_objects as go

from patsy import dmatrices
from scipy.interpolate import interp1d
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from sklearn.tree import DecisionTreeClassifier
```

[3]:
```
# Define general parameters
def _setup():
    input = {
    "startyear" : 1979,
    "endyear" : 2019,
```

```
    "cpi_baseyear" : 2016,
    }
    return input
```

1. Data cleaning and data preparation

1.1 Create a function that cleans `forbalance` data: A raw dataset that contains minimum wage data per state and quarter level.

[4]:
```
def get_forbalance_data(data_forbalance, quarter_codes):
    # Generate the variable 'year' and restrict the dataset to the study period
    data_forbalance["quarterdate"] = pd.
 ↪PeriodIndex(data_forbalance["quarterdate"], freq="Q")
    data_forbalance = pd.merge(data_forbalance, quarter_codes, how="left",␣
 ↪on=["quarterdate"])
    data_forbalance["quarterdate"] = data_forbalance["quarterdate"].astype(str)
    data_forbalance["year"] = pd.to_datetime(data_forbalance["quarterdate"]).dt.
 ↪year
    data_forbalance = data_forbalance.loc[data_forbalance["year"] >=␣
 ↪_setup()["startyear"]]
    return data_forbalance
```

1.2 Create a function that generates `prewindow` data: A data frame with the relevant post and pre-periods around a minimum wage change.

[5]:
```
def get_prewindow_data(data_forbalance, data_eventclass, quarter_codes,␣
 ↪state_codes):
    # Generate the variable 'year' and restrict the dataset to the study period
    data_eventclass = data_eventclass.rename(columns={"quarterdate":␣
 ↪"quarternum"})
    data_eventclass["quarternum"] = data_eventclass["quarternum"].astype(int)
    data_eventclass = pd.merge(data_eventclass, quarter_codes, how="left",␣
 ↪on=["quarternum"])
    data_eventclass["quarterdate"] = data_eventclass["quarterdate"].astype(str)
    data_eventclass["year"] = pd.to_datetime(data_eventclass["quarterdate"]).dt.
 ↪year
    data_eventclass = data_eventclass.loc[data_eventclass["year"] >=␣
 ↪_setup()["startyear"]]

    # Merge `data_eventclass` with `data_forbalance`
    data_forb_event = pd.merge(data_forbalance, data_eventclass, how="left")
    data_forb_event["overallcountgroup"] = data_forb_event["overallcountgroup"].
 ↪fillna(0    )
    data_forb_event.loc[data_forb_event["fedincrease"] == 1,␣
 ↪"overallcountgroup"] = 0
```

```
    data_forb_event.loc[data_forb_event["overallcountgroup"].notna() &␣
↪data_forb_event["overallcountgroup"]> 1,"overallcountgroup"] = 1
    data_forb_event["prewindow"] = 0
    data_forb_event["postwindow"] = data_forb_event["overallcountgroup"]

    for i in range(1, 13):
        data_forb_event[f"L{i}overallcountgroup"] = data_forb_event.
↪groupby(["statenum"])["overallcountgroup"].shift(i, fill_value=0)
        data_forb_event[f"F{i}overallcountgroup"] = data_forb_event.
↪groupby(["statenum"])["overallcountgroup"].shift(-i, fill_value=0)
        data_forb_event["prewindow"] = (data_forb_event["prewindow"] +␣
↪data_forb_event[f"F{i}overallcountgroup"])
        data_forb_event["postwindow"] = (data_forb_event["postwindow"] +␣
↪data_forb_event[f"L{i}overallcountgroup"])

    for i in range(13, 20):
        data_forb_event[f"L{i}overallcountgroup"] = data_forb_event.
↪groupby(["statenum"])["overallcountgroup"].shift(i, fill_value=0)
        data_forb_event["postwindow"] = (data_forb_event["postwindow"] +␣
↪data_forb_event[f"L{i}overallcountgroup"])

    data_forb_event.loc[data_forb_event["postwindow"] >= 1, "postwindow"] = 1
    data_forb_event.loc[data_forb_event["prewindow"] >= 1, "prewindow"] = 1
    data_forb_event.loc[data_forb_event["postwindow"] == 1, "prewindow"] = 0
    prewindow = data_forb_event[["statenum", "quarterdate", "prewindow",␣
↪"postwindow"]]
    prewindow = pd.merge(prewindow, state_codes, how="left", on=["statenum"])
    return prewindow
```

1.3 Create a function that generates `totpop` data: A dataset containing total population variable (number of surveyed people) at state and quarter level.

[6]:
```
# Merge the Current Population Survey (CPS)-ORG and Consumer Price Index (CPI)␣
↪data per month

def _clean_cps_cpi_data(data_cps, data_cpi, state_codes, quarter_codes,␣
↪month_codes):
    # Clean CPI data
    data_cpi = data_cpi.melt(id_vars="year")
    data_cpi = data_cpi.rename(columns={"variable": "monthnum", "value": "cpi",␣
↪"month": "monthnum"})
    data_cpi = data_cpi.loc[data_cpi["year"].
↪between(_setup()["startyear"],_setup()["endyear"])]
    data_cpi["monthnum"] = data_cpi["monthnum"].astype("category")
    data_cpi = pd.merge(data_cpi, month_codes, how="left")
    cpibase = data_cpi.loc[data_cpi["year"] == _setup()["cpi_baseyear"], "cpi"].
↪mean()
```

```
    data_cpi["cpi"] = 100 * (data_cpi["cpi"]/cpibase)

    # Clean the current population survey (CPS)-ORG data
    data_cps.loc[:, "rowid"] = range(1, len(data_cps) + 1)
    data_cps =␣
↪data_cps[['hhid','hhnum','lineno','minsamp','month','state','age','marital',
                        ␣
↪'race','sex','esr','ethnic','uhours','earnhr','uearnwk','earnwt',
                       ␣
↪'uhourse','paidhre','earnhre','earnwke','I25a','I25b','I25c','I25d',
                       ␣
↪'year','lfsr89','lfsr94','statenum','monthdate','quarterdate',
                       ␣
↪'quarter','division','gradeat','gradecp','ihigrdc','grade92','class',
                        ␣
↪'class94','smsa70','smsa80','smsa93','smsa04','smsastat',
                       ␣
↪'prcitshp','penatvty','veteran','pemntvty','pefntvty','vet1','rowid',
                       ␣
↪'ind70','ind80','ind02','occ70','occ80','occ802','occurnum','occ00',
                        'occ002','occ2011','occ2012']]
    data_cps = data_cps.rename(columns={"veteran": "veteran_old","quarterdate":␣
↪"quarternum","month": "month_num"})
    data_cps = data_cps.loc[data_cps["year"] >= _setup()["startyear"]]

    # Merge the current population survey (CPS)-ORG and CPI data
    data_merge_cps_cpi = pd.merge(data_cps, data_cpi, how="left", on=["year",␣
↪"month_num", "monthdate"])
    data_merge_cps_cpi = pd.merge(data_merge_cps_cpi, state_codes, how="left",␣
↪on=["statenum"])
    data_merge_cps_cpi = pd.merge(data_merge_cps_cpi, quarter_codes, how="left",␣
↪on=["quarternum"])
    data_merge_cps_cpi["quarterdate"] = data_merge_cps_cpi["quarterdate"].
↪astype(str)
    return data_merge_cps_cpi

# Generate the totpop data
def get_totpop_data(data_merge_cps_cpi):
    totpop_temp = data_merge_cps_cpi[["monthdate", "quarterdate", "statenum",␣
↪"earnwt"]]
    totpop_temp["totalpopulation"] = totpop_temp.groupby(["statenum",␣
↪"monthdate"], as_index=False)["earnwt"].transform("sum")
    totpop_temp = totpop_temp[["statenum", "quarterdate", "totalpopulation"]]
    totpop = totpop_temp.groupby(["statenum", "quarterdate"], as_index=False).
↪mean()
    return totpop
```

1.4 Create a function that generates `data_cps_cpi` data: A dataset containing hourly wages, weekly earnings, and a range of demographic variables.

[7]:
```
def get_cps_cpi_data(data_merge_cps_cpi):

    # Build variables for imputed hourly wages, imputed weekly earnings, and␣
↪imputed hours worked.
    # Later, observations with imputed hourly wages, imputed weekly earnings, or␣
↪imputed hours worked will be excluded.
    data_cps_cpi = data_merge_cps_cpi.copy()
    data_cps_cpi.loc[data_cps_cpi["paidhre"] == 1, "wage"] =␣
↪(data_cps_cpi["earnhre"] / 100)
    data_cps_cpi.loc[data_cps_cpi["paidhre"] == 2, "wage"] =␣
↪(data_cps_cpi["earnwke"] / data_cps_cpi["uhourse"])
    data_cps_cpi.loc[(data_cps_cpi["paidhre"] == 2) & (data_cps_cpi["uhourse"]␣
↪== 0), "wage"] = np.nan
    data_cps_cpi["hoursimputed"] = np.where((data_cps_cpi["I25a"].notna()) &␣
↪(data_cps_cpi["I25a"] > 0), 1, 0)
    data_cps_cpi["wageimputed"] = np.where((data_cps_cpi["I25c"].notna()) &␣
↪(data_cps_cpi["I25c"] > 0), 1, 0)
    data_cps_cpi["earningsimputed"] = np.where((data_cps_cpi["I25d"].notna()) &␣
↪(data_cps_cpi["I25d"] > 0), 1, 0)

    varlist = ["hoursimputed", "earningsimputed", "wageimputed"]
    for column in data_cps_cpi[varlist]:
        data_cps_cpi.loc[data_cps_cpi["year"].isin(range(1989, 1994)), column] =␣
↪0
        data_cps_cpi.loc[(data_cps_cpi["year"] == 1994) |
            ((data_cps_cpi["year"] == 1995) & (data_cps_cpi["month_num"] <=␣
↪8)),column] = 0

    data_cps_cpi.loc[(data_cps_cpi["year"].isin(range(1989, 1994))) &␣
↪(data_cps_cpi["earnhr"].isin([np.nan, 0]))
        & ((data_cps_cpi["earnhre"].notna()) & (data_cps_cpi["earnhre"] >␣
↪0)),"wageimputed"] = 1
    data_cps_cpi.loc[(data_cps_cpi["year"].isin(range(1989, 1994))) &␣
↪(data_cps_cpi["uhours"].isin([np.nan, 0]))
        & (data_cps_cpi["uhourse"].notna() & (data_cps_cpi["uhourse"] >␣
↪0)),"hoursimputed"] = 1
    data_cps_cpi.loc[(data_cps_cpi["year"].isin(range(1989, 1994))) &␣
↪(data_cps_cpi["uearnwk"].isin([np.nan, 0]))
        & (data_cps_cpi["earnwke"].notna() & (data_cps_cpi["earnwke"] >␣
↪0)),"earningsimputed"] = 1

    data_cps_cpi["imputed"] = np.where((data_cps_cpi["paidhre"] == 2) &
```

```
        ((data_cps_cpi["hoursimputed"] == 1) | (data_cps_cpi["earningsimputed"]␣
↪== 1)), 1, 0)
    data_cps_cpi.loc[(data_cps_cpi["paidhre"] == 1) &␣
↪(data_cps_cpi["wageimputed"] == 1), "imputed"] = 1
    data_cps_cpi.loc[data_cps_cpi["imputed"] == 1, "wage"] = np.nan
    data_cps_cpi["logwage"] = np.where(data_cps_cpi["wage"].notna() &␣
↪(data_cps_cpi["wage"] != 0),np.log(data_cps_cpi["wage"]),np.nan)
    data_cps_cpi["origin_wage"] = data_cps_cpi["wage"]
    data_cps_cpi["wage"] = ((data_cps_cpi["origin_wage"]) / (data_cps_cpi["cpi"]␣
↪/ 100)) * 100
    data_cps_cpi.loc[data_cps_cpi["cpi"] == 0, "wage"] = np.nan
    data_cps_cpi["mlr"] = np.where(data_cps_cpi["year"].isin(range(1979, 1989)),␣
↪data_cps_cpi["esr"], np.nan)
    data_cps_cpi.loc[data_cps_cpi["year"].isin(range(1989, 1994)),"mlr"] =␣
↪data_cps_cpi["lfsr89"]
    data_cps_cpi.loc[data_cps_cpi["year"].isin(range(1994, 2020)),"mlr"] =␣
↪data_cps_cpi["lfsr94"]

    # Build demographic variables
    data_cps_cpi["hispanic"] = np.where((data_cps_cpi["year"].isin(range(1976,␣
↪2003))) & (data_cps_cpi["ethnic"].isin(range(1, 8))),1, 0)
    data_cps_cpi.loc[(data_cps_cpi["year"].isin(range(2003, 2014))) &␣
↪(data_cps_cpi["ethnic"].isin(range(1, 6))),"hispanic"] = 1
    data_cps_cpi.loc[(data_cps_cpi["year"].isin(range(2014, 2020))) &␣
↪(data_cps_cpi["ethnic"].isin(range(1, 10))),"hispanic"] = 1

    data_cps_cpi["black"] = np.where((data_cps_cpi["race"] == 2) &␣
↪(data_cps_cpi["hispanic"] == 0), 1, 0)
    data_cps_cpi.loc[data_cps_cpi["race"] >= 2,"race"] = 2

    data_cps_cpi["dmarried"] = np.where(data_cps_cpi["marital"] <= 2, 1, 0)
    data_cps_cpi.loc[data_cps_cpi["marital"].isna(),"dmarried"] = np.nan

    data_cps_cpi["sex"] = data_cps_cpi["sex"].replace(2, 0)

    data_cps_cpi["hgradecp"] = np.where(data_cps_cpi["gradecp"] == 1,␣
↪data_cps_cpi["gradeat"], np.nan)
    data_cps_cpi["hgradecp"] = np.where(data_cps_cpi["gradecp"] == 2,␣
↪data_cps_cpi["gradeat"] - 1, data_cps_cpi["hgradecp"])
    data_cps_cpi.loc[data_cps_cpi["ihigrdc"].notna() & data_cps_cpi["hgradecp"].
↪isna(),"hgradecp"] = data_cps_cpi["ihigrdc"]
    grade92code = list(range(31, 47))
    impute92code = (0, 2.5, 5.5, 7.5, 9, 10, 11, 12, 12, 13, 14, 14, 16, 18, 18,␣
↪18)
    for i, j in zip(grade92code, impute92code):
        a = i
```

```
        b = j
        data_cps_cpi.loc[data_cps_cpi["grade92"] == a, "hgradecp"] = b
    data_cps_cpi["hgradecp"] = data_cps_cpi["hgradecp"].replace(-1, 0)
    data_cps_cpi["hsl"] = np.where(data_cps_cpi["hgradecp"] <= 12, 1, 0)
    data_cps_cpi["hsd"] = np.where(
        (data_cps_cpi["hgradecp"] < 12) & (data_cps_cpi["year"] < 1992), 1, 0
    )
    data_cps_cpi.loc[(data_cps_cpi["grade92"] <= 38) & (data_cps_cpi["year"] >=␣
↪1992),"hsd"] = 1
    data_cps_cpi["hs12"] = np.where((data_cps_cpi["hsl"] == 1) &␣
↪(data_cps_cpi["hsd"] == 0), 1, 0)
    data_cps_cpi["conshours"] = np.where(data_cps_cpi["I25a"] == 0,␣
↪data_cps_cpi["uhourse"], np.nan)
    data_cps_cpi["hsl40"] = np.where((data_cps_cpi["hsl"] == 1) &␣
↪(data_cps_cpi["age"] < 40), 1, 0)
    data_cps_cpi["hsd40"] = np.where((data_cps_cpi["hsd"] == 1) &␣
↪(data_cps_cpi["age"] < 40), 1, 0)
    data_cps_cpi["sc"] = np.where(data_cps_cpi["hgradecp"].isin([13, 14, 15]) &␣
↪(data_cps_cpi["year"] < 1992),1,0)
    data_cps_cpi.loc[data_cps_cpi["grade92"].isin(range(40, 43)) &␣
↪(data_cps_cpi["year"] >= 1992),"sc"] = 1
    data_cps_cpi["coll"] = np.where((data_cps_cpi["hgradecp"] > 15) &␣
↪(data_cps_cpi["year"] < 1992), 1, 0)
    data_cps_cpi.loc[(data_cps_cpi["grade92"] > 42) & (data_cps_cpi["year"] >=␣
↪1992),"coll"] = 1

    data_cps_cpi["ruralstatus"] = np.where(data_cps_cpi["smsastat"] == 2, 1, 2)

    data_cps_cpi = data_cps_cpi.drop(['smsastat','smsa80','smsa93','smsa04'],␣
↪axis=1)

    data_cps_cpi["veteran"] = np.where(data_cps_cpi["veteran_old"].notna() &␣
↪(data_cps_cpi["veteran_old"] != 6), 1, 0)
    data_cps_cpi.loc[data_cps_cpi["vet1"].notna(),"veteran"] = 1

    data_cps_cpi["educcat"] = np.where(data_cps_cpi["hsd"] == 1, 1, 0)
    data_cps_cpi.loc[data_cps_cpi["hs12"] == 1,"educcat"] = 2
    data_cps_cpi.loc[data_cps_cpi["sc"] == 1,"educcat"] = 3
    data_cps_cpi.loc[data_cps_cpi["coll"] == 1,"educcat"] = 4

    data_cps_cpi["agecat"] = pd.cut(data_cps_cpi["age"], bins=[0, 20, 25, 30,␣
↪35, 40, 45, 50, 55, 60, 100])
    return data_cps_cpi
```

1.5 Compile the earlier functions in one function that generates the dataset relevant for prediction purposes.

[8]:
```
def get_forprediction_eventstudy_data(data_forbalance, data_eventclass,␣
↪data_cps, data_cpi,
                                      quarter_codes, state_codes, month_codes):
    """
    Args:
        data_forbalance (pd.DataFrame): A raw dataset that
            contains minimum wage data per state and quarter level.
        data_eventclass (pd.DataFrame): A raw dataset with information
            to identify the relevant post and pre-period around prominent␣
↪minimum wage changes.
        data_cps (pd.DataFrame): The raw Current Population Survey (CPS)-ORG␣
↪dataset containing hourly wages,
            weekly earnings, and a range of demographic variables.
        data_cpi (pd.DataFrame): The raw Consumer Price Index (CPI) dataset per␣
↪month.
        state_codes (pd.DataFrame):
            A dataframe containing state information.
        quarter_codes (pd.DataFrame):
            A dataframe containing quarter information.
        month_codes (pd.DataFrame):
            A dataframe containing month information.
    """
    forbalance = get_forbalance_data(data_forbalance, quarter_codes)
    prewindow = get_prewindow_data(forbalance, data_eventclass, quarter_codes,␣
↪state_codes)
    data_merge_cps_cpi = _clean_cps_cpi_data(data_cps, data_cpi, state_codes,␣
↪quarter_codes, month_codes)
    totpop = get_totpop_data(data_merge_cps_cpi)
    data_cps_cpi = get_cps_cpi_data(data_merge_cps_cpi)
    merge_1 = pd.merge(data_cps_cpi, totpop, how="inner", on=["statenum",␣
↪"quarterdate"])
    merge_2 = pd.merge(merge_1,forbalance,how="inner",on=["statenum",␣
↪"quarterdate", "year", "quarternum"])
    merge_3 = pd.merge(merge_2, prewindow, how="inner", on=["statenum",␣
↪"quarterdate","state_name"])
    merge_3["MW"] = np.exp(merge_3["logmw"])
    merge_3["ratio_mw"] = merge_3["origin_wage"] / merge_3["MW"]
    merge_3.loc[merge_3["MW"] == 0, "ratio_mw"] = 0
    merge_3.loc[(merge_3["ratio_mw"] < 1) & (merge_3["origin_wage"].notna()) &␣
↪(merge_3["origin_wage"] > 0),"relMW_groups"] = 1
    merge_3.loc[(merge_3["ratio_mw"] >= 1) & (merge_3["ratio_mw"] < 1.25) &␣
↪(merge_3["origin_wage"].notnull()),"relMW_groups"] = 2
    merge_3.loc[(merge_3["ratio_mw"] >= 1.25) & (merge_3["origin_wage"].
↪notna()), "relMW_groups"] = 3
    merge_3["training"] = np.where((merge_3["prewindow"] == 1) &␣
↪(~merge_3["origin_wage"].isin([0, np.nan])), 1, 0)
```

```
    merge_3["validation"] = np.where((merge_3["prewindow"] == 0) &␣
↪(merge_3["postwindow"] == 0) & (~merge_3["origin_wage"].isin([0, np.nan])),1,0)
    return merge_3
```

[9]:
```
    url_state_codes = "https://www.dropbox.com/s/5ib83ob02h5119j/state_codes.pkl?
↪dl=1"
    url_quarter_codes = "https://www.dropbox.com/s/pnc2u5in19izihg/quarter_codes.
↪pkl?dl=1"
    url_month_codes = "https://www.dropbox.com/s/1b1ro2xp7prbqwi/month_codes.pkl?
↪dl=1"
    url_data_forbalance = "https://www.dropbox.com/s/jlpjjjc0youx1hi/
↪VZmw_quarterly_lagsleads_1979_2019.dta?dl=1"
    url_data_eventclass = "https://www.dropbox.com/s/p5barkrnh8yhcxh/
↪eventclassification_2019.dta?dl=1"
    url_data_cpi = "https://www.dropbox.com/s/0mjybcm9h9pj54y/cpiursai1977-2019.
↪dta?dl=1"
    url_data_cps_morg = "https://www.dropbox.com/s/xcztuhcmorxwq9b/
↪cps_morg_2019_new.dta?dl=1"

    state_codes = pd.read_pickle(url_state_codes)
    quarter_codes = pd.read_pickle(url_quarter_codes)
    month_codes = pd.read_pickle(url_month_codes)
    data_forbalance = pd.read_stata(url_data_forbalance)
    data_eventclass = pd.read_stata(url_data_eventclass,␣
↪convert_categoricals=False)
    data_cpi = pd.read_stata(url_data_cpi)
    data_cps_morg = pd.read_stata(url_data_cps_morg, convert_categoricals=False)
```

[10]:
```
forprediction_eventstudy =␣
↪get_forprediction_eventstudy_data(data_forbalance,data_eventclass,
                                                                        ␣
↪data_cps_morg,data_cpi,quarter_codes,
                                                                        ␣
↪state_codes,month_codes)
```

[11]:
```
def get_fortraining_eventstudy_data(data):
    """Generates a dataset for prediction excluding self-employed.
    """
    fortraining = data[data["relMW_groups"].notna()]
    fortraining = fortraining[~(fortraining["class"].isin([5, 6]) &␣
↪(fortraining["year"] <= 1993))]
    fortraining = fortraining[~(fortraining["class94"].isin([6, 7]) &␣
↪(fortraining["year"] >= 1994))]
    return fortraining
```

```
[25]: fortraining_eventstudy =␣
       ↪get_fortraining_eventstudy_data(forprediction_eventstudy)
```

2. Predicting who is a minimum wage worker

2.1. Create a function to split the full data into training a testing datasets.

```
[13]: def _get_fortraining_data(data):
          # data: The full data set generated as a result of the data cleaning part.
          data["relMW_groups"] = np.where(data["relMW_groups"] != 3, 1, 0)
          cat_cols =␣
       ↪["ruralstatus","sex","hispanic","dmarried","race","veteran","educcat","agecat"]
          data[cat_cols] = data[cat_cols].astype("category")

          data_full = data.loc[(data["training"] == 1) & (~data["quarterdate"].
       ↪isin(range(136, 143)))]
          data_full =␣
       ↪data_full[["race","sex","hispanic","agecat","age","dmarried","educcat","relMW_groups","rurals

          x_train, x_test, y_train, y_test = train_test_split(data_full.
       ↪drop(["relMW_groups"], axis=1),
              data_full["relMW_groups"], test_size=0.3, random_state=12345)

          data_train = pd.concat([y_train,x_train], axis=1)
          data_test = pd.concat([y_test,x_test], axis=1)
          return (data_full, data_train, data_test)
```

2.2 Training machine learning models to predict individual's exposure to a minimum wage change

- Create a function that trains the decision tree model

```
[14]: def _pred_decision_tree(data_train,data_test):
          formula =␣
       ↪"relMW_groups~age+race+sex+hispanic+dmarried+ruralstatus+educcat+veteran"
          y_tr,x_tr = dmatrices(formula,data_train,return_type="dataframe")
          _,x_ts = dmatrices(formula,data_test,return_type="dataframe")

          tree_income = DecisionTreeClassifier().fit(x_tr, y_tr)
          yhat_tree = tree_income.predict_proba(x_ts)[:, 1]
          return yhat_tree
```

- Create a function that trains the gradient-boosting tree model

```
[15]: def _pred_boosted_tree(data_train, data_test):
          formula =␣
       ↪"relMW_groups~age+race+sex+hispanic+dmarried+ruralstatus+educcat+veteran"
          y_tr,x_tr = dmatrices(formula,data_train,return_type="dataframe")
          _,x_ts = dmatrices(formula,data_test,return_type="dataframe")
```

```
    boost_income = GradientBoostingClassifier(n_estimators=4000,
                                              learning_rate=0.005,
                                              max_depth=6,
                                              min_samples_leaf = 10).fit(x_tr,␣
↪y_tr)
    yhat_boost = boost_income.predict_proba(x_ts)[:, 1]
    return yhat_boost
```

- Create a function that trains a random forest model

[16]:
```
def _pred_random_forest(data_train, data_test):
    formula =␣
↪("relMW_groups~age+race+sex+hispanic+dmarried+ruralstatus+educcat+veteran")
    y_tr,x_tr = dmatrices(formula,data_train,return_type="dataframe")
    y_ts,x_ts = dmatrices(formula,data_test,return_type="dataframe")

    rf_income = RandomForestClassifier(n_estimators=2000, max_features=2).fit(
        x_tr, y_tr
    )
    yhat_rf = rf_income.predict_proba(x_ts)[:, 1]
    return yhat_rf
```

- Create a function that fits the linear Card and Krueger probability model

[17]:
```
def _pred_linear_model(data_train, data_test):
    formula = "relMW_groups2 ~␣
↪age+hispanic+race+sex+educcat_2+educcat_3+educcat_4"
    formula = " + ".join([formula] + [f"I(age ** {i})" for i in range(2, 4)] +␣
↪["race2:sex:teen"] + ["race2:sex:young_adult"] + [f"age:sex:educcat_{i}" for i␣
↪in range(2,5)])
    lm_fit = smf.ols(formula, data=data_train).fit()
    yhat_lm = lm_fit.predict(data_test)
    return yhat_lm
```

- Create a function that fits a basic logistic model

[18]:
```
def _pred_basic_logit(data_train, data_test):
    formula = "relMW_groups ~ age+ educcat_2 + educcat_3 + educcat_4"
    logit_fit = smf.logit(
        formula, data=data_train
    ).fit()
    log_odds = logit_fit.predict(data_test)
    yhat_logit = 1 / (1 + np.exp(-log_odds))
    return yhat_logit
```

- Compile the earlier functions in one function and compute precision and recall values (relevant to compare later the performance of the prediction models)

[19]:
```python
def get_precision_recall(data):
    data_full, data_train, data_test = _get_fortraining_data(data)
    yhat_boost = _pred_boosted_tree(data_train, data_test)
    yhat_tree = _pred_decision_tree(data_full,data_test)
    yhat_rf = _pred_random_forest(data_train, data_test)

    dfs = [data_train, data_test]
    for data in dfs:
        data["teen"] = np.where(data["age"] < 20, 1, 0)
        data["race2"] = np.where(data["race"] == 1, 1, 0)
        data["young_adult"] = np.where(data["age"].isin(range(20, 26)), 1, 0)
        data["relMW_groups2"] = data["relMW_groups"].astype(int) - 1

    data_train = pd.get_dummies(data_train, columns=['educcat'])
    data_test = pd.get_dummies(data_test, columns=['educcat'])

    yhat_lm = _pred_linear_model(data_train, data_test)
    yhat_logit = _pred_basic_logit(data_train, data_test)

    precision_dict = {}
    recall_dict = {}
    f_interp = {}
    precision_df = {}

    (precision_df["precision_boost"],precision_df["recall_boost"],_) =␣
↪precision_recall_curve(data_test["relMW_groups"], yhat_boost)
    var_names = ["rf", "tree", "lm", "logit"]
    ytest = data_test["relMW_groups"]

    for name in var_names:
        yhat = vars()[f"yhat_{name}"]
        (precision_dict[f"precision_{name}"],recall_dict[f"recall_{name}"],_) =␣
↪precision_recall_curve(ytest, yhat)
        f_interp[f"f_interp_{name}"] = interp1d(recall_dict[f"recall_{name}"],␣
↪precision_dict[f"precision_{name}"])
        precision_df[f"precision_interp_{name}"] =␣
↪f_interp[f"f_interp_{name}"](precision_df["recall_boost"])
        precision_df[f"precision_diff_{name}"] =␣
↪(precision_df[f"precision_interp_{name}"] - precision_df["precision_boost"])

    return precision_df
```

[20]:
```python
precision_df = get_precision_recall(fortraining_eventstudy)
precision = pd.DataFrame.from_dict(precision_df)
```

```
C:\Users\lucia\AppData\Roaming\Python\Python39\site-
packages\sklearn\ensemble\_gb.py:424: DataConversionWarning: A column-vector y
```

```
was passed when a 1d array was expected. Please change the shape of y to
(n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
C:\Users\lucia\AppData\Roaming\Python\Python39\site-
packages\sklearn\base.py:1152: DataConversionWarning: A column-vector y was
passed when a 1d array was expected. Please change the shape of y to
(n_samples,), for example using ravel().
  return fit_method(estimator, *args, **kwargs)
```

```
Optimization terminated successfully.
         Current function value: 0.321767
         Iterations 7
```

2.3 Compare the performance of the prediction models

- Plot the precision-recall curves for all the prediction models trained earlier

[21]:
```python
def plot_precision_recall_curve(data):
    # data: A DataFrame containing the recall and precision values for each␣
 ↪model to be plotted.
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=data['recall_boost'], y=data['precision_boost'],␣
 ↪mode='lines', name='Boosted Trees'))
    fig.add_trace(go.Scatter(x=data['recall_boost'],␣
 ↪y=data['precision_interp_rf'], mode='lines', name='Random Forest'))
    fig.add_trace(go.Scatter(x=data['recall_boost'],␣
 ↪y=data['precision_interp_tree'], mode='lines', name='Tree'))
    fig.add_trace(go.Scatter(x=data['recall_boost'],␣
 ↪y=data['precision_interp_lm'], mode='lines', name='Linear (Card & Krueger)'))
    fig.add_trace(go.Scatter(x=data['recall_boost'],␣
 ↪y=data['precision_interp_logit'], mode='lines', name='Basic Logistic'))
    fig.update_layout(title='Precision-Recall Curve',
                      xaxis_title='Recall',
                      yaxis_title='Precision')
    return fig
```

[22]:
```python
plot_precision_recall_curve(precision)
```

2.4 Who are the minimum wage workers?

- Plot the feature importance i.e. relative influences of the predictors in the gradient-boosting tree prediction model.

[23]:
```python
def get_boost_basic_model(data):
    # Train the basic boosted tree model.
    # data (pd.DataFrame): The full data set generated as a result of the data␣
 ↪cleaning part.
    data_train = _get_fortraining_data(data)[1]
```

```
    xtr_basic,ytr_basic =␣
↪data_train[["age","race","sex","hispanic","dmarried","ruralstatus","educcat","veteran"]],data
    boost_basic = GradientBoostingClassifier(n_estimators=4000, learning_rate=0.
↪005, max_depth=6, min_samples_leaf = 10).fit(xtr_basic,ytr_basic)
    return boost_basic

def feature_importance(boost_basic_model):
    feature_labels =␣
↪["age","race","sex","hispanic","dmarried","ruralstatus","educcat","veteran"]
    feature_importance = boost_basic_model.feature_importances_
    data = {"feature": feature_labels, "importance": feature_importance}
    data = pd.DataFrame(data)
    ind = np.argsort(data["importance"], axis=1)
    sorted_labels = data["importance"][ind]
    sorted_importance = data["feature"][ind]
    fig = go.Figure(data=[go.Bar(x=sorted_labels, y=sorted_importance,␣
↪orientation='h')])
    fig.update_layout(title='Feature Importance',
                      yaxis_title='Feature',
                      xaxis_title='Importance')
    return fig
```

[26]:
```
boost_basic_model = get_boost_basic_model(fortraining_eventstudy)
feature_importance(boost_basic_model)
```