Problem Set 3 - Andreas Bloch

Imports

First, we'll import a set of functions that we'll use over and over again.

In [77]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import os
import numpy as np
import scipy
import pandas as pd
import ggplot as gg
import itertools
from collections import Counter
import random
from gensim.models import Word2Vec
from gensim.models.callbacks import CallbackAny2Vec
from sklearn.manifold import TSNE
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.model_selection import KFold
from math import isnan
from zipfile import ZipFile
import spacy
from spacy.lang.en.stop_words import STOP_WORDS
from spacy.lemmatizer import Lemmatizer
from spacy.lang.en import LEMMA_INDEX, LEMMA_EXC, LEMMA_RULES
from nltk import ngrams
from nltk.stem import SnowballStemmer
from string import punctuation
import re
from tqdm import tqdm
```

Data Preprocessing

Here we'll define the functionality to preprocesses the data for exercises 1 and 2

In [2]:

```
DATA_DIR = './data'
CASE_METADATA_FILENAME = 'case_metadata.csv'

class CounterMessage(object):

    def __init__(self):
        self.counter = 1
        self.last_action = ''

    def update(self, max_cnt, action):
        # reset counter for every new action
        if self.last_action != action:
            self.counter = 1
            self.last_action = action
            print('')
        # print status message
        print('\r{}/{}: {}.'.format(self.counter, max_cnt, action),
              end=' ' * 10, flush=True)
        self.counter += 1

def create_exercise_1_2_dataset():

    # create message counter
    counter = CounterMessage()

    # open cases zip file
    zfile = ZipFile('data/cases.zip')
    caseids = []
    raw_texts = {}

    # randomly shuffle files
    members = zfile.namelist()
    NUM_CASES = len(members)

    for case in members:
        year, caseid = case[:-4].split('_')
        with zfile.open(case) as f:
            raw_text = f.read().decode()
        raw_texts[caseid] = raw_text
        caseids.append(caseid)
        counter.update(NUM_CASES, 'opened')

    # do NLP
    nlp = spacy.load('en')
    spacy_documents = {}
    for caseid in caseids:
        spacy_documents[caseid] = nlp(raw_texts[caseid])
        counter.update(NUM_CASES, 'nlp-processed')

    # create punctuation remover
    punctuation_remover = str.maketrans('', '', punctuation)
    # create lemmatizer
    lemmatizer = Lemmatizer(LEMMA_INDEX, LEMMA_EXC, LEMMA_RULES)

    def filter_and_transform(token):
        # get the token's word(s)
        word = ''.join(token.text)
        # replace newlines with spaces
        word = word.replace('\r', ' ').replace('\n', ' ')
```

```
    # remove punctuation
    word = word.translate(punctuation_remover)
    # replace multiple subsequent spaces with one space
    word = re.sub(' +', ' ', word)
    # check that word still has some text (not just one char or space)
    if len(word) <= 1:
        return False, (word, token.pos_)
    # normalize numbers (28, 28th, 1st, ...)
    if any(char.isdigit() for char in word):
        return False, (word, token.pos_)
    # lemmatize the word
    lemmas = lemmatizer(word, token.pos)[0]
    # try to lemmatize the word
    if isinstance(lemmas, (list,)) and len(lemmas) > 0:
        # pick the first option if several lemmas were found
        word = lemmas[0]
    else:
        # no lemma was found (just keep the original word)
        word = word
    # convert the word to lowercase
    word = word.lower()
    # remove stopwords
    if word in STOP_WORDS:
        return False, (word, token.pos_)
    # finally, return the filtered word and type
    return True, (word, token.pos_)


# create filtered texts
case_sentences = {k: [] for k in caseids}

# process sentences of cases
for caseid in caseids:
    spacy_document = spacy_documents[caseid]
    # process each sentence separately
    curr_case_sentences = []
    for sentence in spacy_document.sents:
        # create a list of tokens for each sentence
        sentence_tokens = []
        # filter the tokens in the case
        for token in sentence:
            take_token, filtered_token = filter_and_transform(token)
            if take_token:
                sentence_tokens.append(filtered_token[0])
        # append list of processed tokens for this document
        if len(sentence_tokens) > 0:
            curr_case_sentences.append(sentence_tokens)
    # save list of all sentences of this case
    if len(curr_case_sentences) > 0:
        case_sentences[caseid] = curr_case_sentences
    else:
        print('case' + str(caseid) + 'has no sentences!')

    counter.update(NUM_CASES, 'sentences filtered.')

# load metadata to get full data
full_data = {}
case_metadata = pd.read_csv(os.path.join(DATA_DIR,
                                         CASE_METADATA_FILENAME)).values
for caseid, case_reversed, judge_id, year,\
    x_republican, log_cites in case_metadata:
```

```
        if caseid in case_sentences and not isnan(x_republican):
            full_data[caseid] = [case_sentences[caseid], x_republican]

    # create data frame
    df = pd.DataFrame.from_dict(full_data,
                                orient='index',
                                columns=['sentences', 'republican'])

    # store data frame
    df.to_pickle('data.pkl')
```

Let's create the preprocessed data for exercises 1 and 2 (will be saved into 'data.pkl')

In [3]:

```
create_exercise_1_2_dataset()
```

```
5762/5762: opened.
5762/5762: nlp-processed.
5762/5762: sentences filtered..
```

Exercise 1

Q: Train a word embedding (Word2Vec, GloVe, ELMo, BERT, etc) on your corpus, once with a small window (e.g. 2) and again with a long window (e.g. 16). What do you expect to change for the different window sizes? Pick a sample of 100 words and visualize them in two dimensions, to demonstrate the difference between the models.

A: First, let's define the functionality that performs a word-embedding (skip-gram) for a set of sentences.

In [4]:

```python
class loss_callback(CallbackAny2Vec):
    '''callback to print loss after each epoch.'''
    def __init__(self):
        self.epoch = 0
        self.last_loss = 0

    def on_epoch_end(self, model):
        cumulative_loss = model.get_latest_training_loss()
        loss = cumulative_loss - self.last_loss
        self.last_loss = cumulative_loss
        print('Epoch %3d, Loss: %12.2f' % (self.epoch, loss))
        self.epoch += 1


def train_w2v(sentences, window):
    """
    Trains and returns a word2vec model (skip-gram).
    :param sentences: the sentences to use for training, e.g.,
    [['it', 'was', ...],...,['today', 'i', 'will', ...]]
    :param window:
    :return: trained word2vec model
    """

    # shuffle sentences (in-place)
    random.shuffle(sentences)

    # create and train word2vec model
    w2v = Word2Vec(
        sentences,          # list of tokenized sentences
        size=300,           # embedding dimensionality
        window=window,      # context window size
        min_count=5,        # only include words with minimum word count
        workers=16,         # num threads (parallelization)
        sg=1,               # use skip-gram model
        negative=20,        # number of negative samples per positive sample
        ns_exponent=3/4,    # damping of frequent words for neg. sampling
        sample=1e-3,        # downsample frequent words up to threshold
        alpha=0.025,        # initial learning rate
        min_alpha=0.0001,   # linearly drop learning rate until
        seed=23,            # random number seed
        iter=20,            # number of iterations over the corpus
        sorted_vocab=1,     # sort the vocabulary by descending frequency
        compute_loss=True,  # compute training loss
        callbacks=[         # callbacks
            loss_callback() # training loss callback (every epoch)
        ]
    )

    # done training, so delete context vectors
    # precompute L2-normalized vectors used for similarities
    # (forget original vectors and keep normalized vectors)
    # mdel becomes read-only and cannot be trained anymore
    w2v.wv.init_sims(replace=True)

    return w2v
```

Further, let's define the to create a 2D-visualization of the embedding via t-SNE.

In [5]:

```
def train_and_get_visualization(w2v, window_size, num_words=200):
    """
    Visualizes a word embedding (instances, dim) in 2 dimensions.
    :param w2v: word-embeddings (instances, dim)
    :param num_words: number of words to visualize (make sure embedding vectors
    are sorted in desired order!) the first 'num_words' vectors will be
    displayed
    :return: None
    """

    # create tsne object
    tsne = TSNE(n_components=2,
                perplexity=50,
                early_exaggeration=12.0,
                learning_rate=200.0,
                n_iter=1000,
                n_iter_without_progress=300,
                metric = scipy.spatial.distance.cosine,
                verbose=1)

    # the words are sorted by their frequencies,
    # so we get the most frequent 'num_words'
    chart_data = pd.DataFrame(list(enumerate(w2v.wv.index2word, 0))[:num_words],
                              columns=['word_index', 'word'])

    # get word embedding vectors
    word_vectors = w2v.wv.vectors[:num_words,:]

    # create 2D representation for word embeddings via TSNE
    wv_tsne = tsne.fit_transform(word_vectors)

    # get x and y coordinates in 2D visualization
    chart_data['x'] = wv_tsne[:, 0]
    chart_data['y'] = wv_tsne[:, 1]

    # create and show chart

    chart = gg.ggplot(chart_data, gg.aes(x='x', y='y', label='word')) + \
            gg.geom_text(size=10, alpha=.8, label='word')
    chart.title = 'Window Size: '+str(window_size)
    return chart
```

This function will be useful to convert the dataframe columns into lists of sentences

In [6]:

```
# flattens a dataframe column (= list of lists of sent.) to list of sent.
# where sent. = list of stemmed words
def flatten(df_col):
    # convert dataframe column to list of lists
    list_of_list_of_sentences = df_col.tolist()
    # flatten list of lists
    L = list(itertools.chain.from_iterable(list_of_list_of_sentences))
    return L
```

Now, let's get to the embedding process:

In [7]:

```
# read preprocessed data frame
df = pd.read_pickle('data.pkl')

# get all sentences
sentences = flatten(df['sentences'])

# train embeddings with window size of 2 and 16
w2v_2 = train_w2v(sentences, window=2)
w2v_16 = train_w2v(sentences, window=16)
```

```
Epoch   0, Loss:   3028862.00
Epoch   1, Loss:   2521678.00
Epoch   2, Loss:   2334689.00
Epoch   3, Loss:   2424485.00
Epoch   4, Loss:   3180235.00
Epoch   5, Loss:   2637016.00
Epoch   6, Loss:   2772657.00
Epoch   7, Loss:   2650442.00
Epoch   8, Loss:   2992980.00
Epoch   9, Loss:   2823214.00
Epoch  10, Loss:   1808112.00
Epoch  11, Loss:   2256428.00
Epoch  12, Loss:   2689286.00
Epoch  13, Loss:   1719448.00
Epoch  14, Loss:   2140580.00
Epoch  15, Loss:   1733912.00
Epoch  16, Loss:   2063840.00
Epoch  17, Loss:   3095780.00
Epoch  18, Loss:   1654688.00
Epoch  19, Loss:   2496572.00
Epoch   0, Loss:  13554064.00
Epoch   1, Loss:   6789278.00
Epoch   2, Loss:   5786298.00
Epoch   3, Loss:   5743660.00
Epoch   4, Loss:  13756968.00
Epoch   5, Loss:   9205584.00
Epoch   6, Loss:   6824356.00
Epoch   7, Loss:   5841960.00
Epoch   8, Loss:   2091560.00
Epoch   9, Loss:   3169936.00
Epoch  10, Loss:   1620256.00
Epoch  11, Loss:   1731360.00
Epoch  12, Loss:   1696608.00
Epoch  13, Loss:   1357392.00
Epoch  14, Loss:   1659608.00
Epoch  15, Loss:   1217816.00
Epoch  16, Loss:   1699944.00
Epoch  17, Loss:   2022344.00
Epoch  18, Loss:   2248064.00
Epoch  19, Loss:   2156688.00
```

Now, let's train the visualizations:

In [8]:

```
# train 2D visualization of embeddings for window sizes of 2 and 16
tsne_2_chart = train_and_get_visualization(w2v_2, 2, num_words=100)
tsne_16_chart = train_and_get_visualization(w2v_16, 16, num_words=100)
```

```
[t-SNE] Computing 99 nearest neighbors...
[t-SNE] Indexed 100 samples in 0.000s...
[t-SNE] Computed neighbors for 100 samples in 0.346s...
[t-SNE] Computed conditional probabilities for sample 100 / 100
[t-SNE] Mean sigma: 0.263339
[t-SNE] KL divergence after 250 iterations with early exaggeration:
51.416550
[t-SNE] KL divergence after 1000 iterations: 0.366523
[t-SNE] Computing 99 nearest neighbors...
[t-SNE] Indexed 100 samples in 0.000s...
[t-SNE] Computed neighbors for 100 samples in 0.343s...
[t-SNE] Computed conditional probabilities for sample 100 / 100
[t-SNE] Mean sigma: 0.285375
[t-SNE] KL divergence after 250 iterations with early exaggeration:
50.988808
[t-SNE] KL divergence after 850 iterations: 0.339841
```

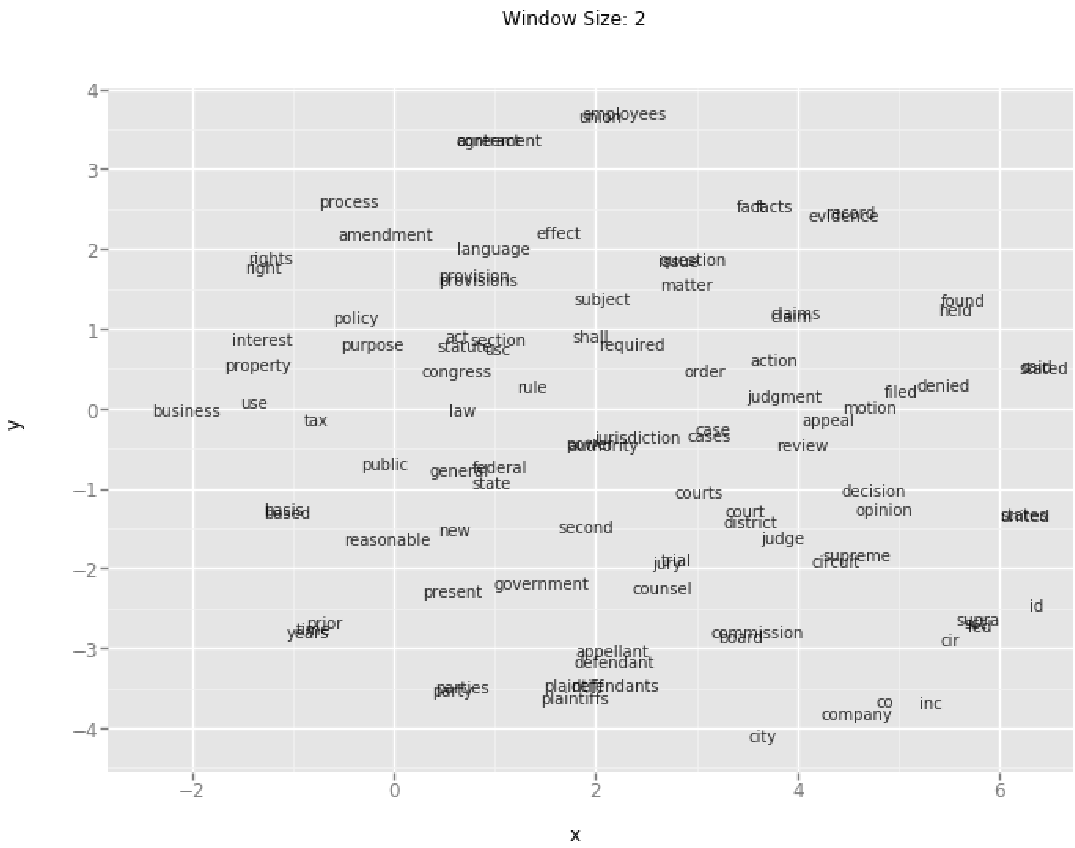
I expect that with the smaller window size words are grouped in a more paradigmatic manner. In this case words are close to their potential substitutes (e.g., dog VS cat). The training of each word is made out of little context, which makes the words more substitutable (but also change the topic). So the substitutes may not necessarily belong to the same topic, or mean the same thing, but they just fit as a substitute for that word in a small context of a sentence.

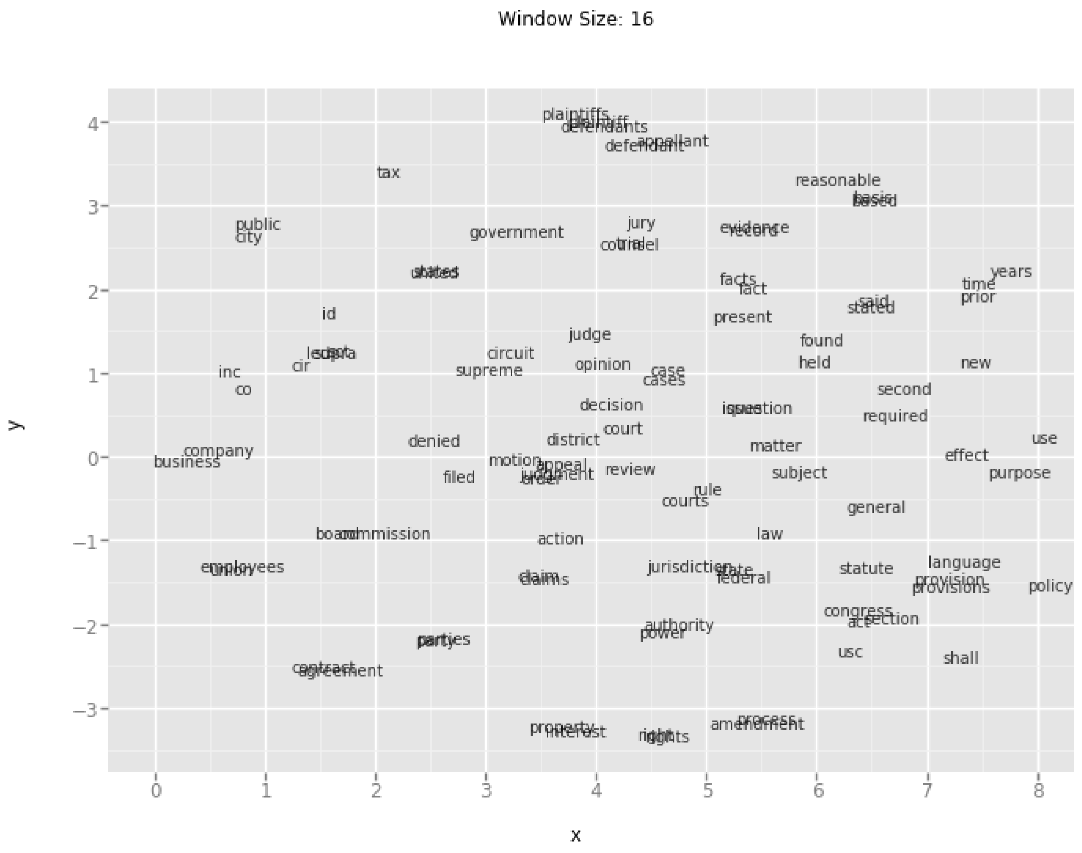
I expect that with a larger window, since in that case every time we're training with a larger context, the model learns better which word exactly fits into which context. So I expect words to be grouped by topic (rather being close to substitutes that could be fit in their place to get a syntactically valid sentence, but likely with a very different meaning). So I expect the words to be grouped in a syntagmatic fashion.

The effect isn't that evident in the results below. The two graphs show the 100 most frequent words embedded in 2D via t-SNE. For example, one can see that in the window size of 16 the words 'business' and 'company' are grouped (as they belong to the same topic), whereas with the window size of 2 the words are apart (company is next to its substitutes 'co', 'inc' etc, far apart from business).

In [9]:

```
# show 2D visualizations
tsne_2_chart.show()
tsne_16_chart.show()
```


Window Size: 2
y
x
employees
process
facts
amendment
effect
language
rights
right
provisions
matter
subject
found
held
policy
claims
interest
purpose
section
shall
required
property
congress
order
action
stated
rule
judgment
filed
denied
business
use
tax
law
motion
appeal
jurisdiction
case
cases
review
public
federal
state
courts
decision
court
district
opinion
reasonable
new
second
judge
supreme
circuit
trial
jury
government
present
counsel
id
prior
years
commission
board
cir
appellant
defendant
defendants
plaintiffs
co
inc
company
city



Exercise 2

Q: Train separate word embeddings for Republican and Democrat judges. Use your word embeddings to list the adjectives most associated with a social group or concept of your choice (following, for example, the method in Caliskan et al 2017 or Kozlowski et al 2018), and analyze differences by judge party.

A: I've used the approach of Kozlowski (mean of pairs) to compute the dimensions of the concepts 'gender', 'class' and 'race' and I've analyzed how various terms of the topics 'jobs' and 'crimes' are placed in the republican and democrat embedding.

First we'll get all the sentences written by republican and democrat judges

In [10]:

```
# read preprocessed data frame
df = pd.read_pickle('data.pkl')

# get all republican and democrat sentences
republican_sentences = flatten(df[df['republican'] > 0.5]['sentences'])
democrat_sentences = flatten(df[df['republican'] < 0.5]['sentences'])

print(len(republican_sentences))
print(len(democrat_sentences))
```

```
375810
392168
```

Now we'll train the corresponding embeddings

In [11]:

```
# train republican and democrat embeddings
w2v_republican = train_w2v(republican_sentences, window=16)
w2v_democrat = train_w2v(democrat_sentences, window=16)
```

```
Epoch   0, Loss:   5511577.00
Epoch   1, Loss:   5866679.00
Epoch   2, Loss:   5211086.00
Epoch   3, Loss:   5401818.00
Epoch   4, Loss:   5183002.00
Epoch   5, Loss:   5121732.00
Epoch   6, Loss:   5411254.00
Epoch   7, Loss:   4560952.00
Epoch   8, Loss:   7606512.00
Epoch   9, Loss:   3920756.00
Epoch  10, Loss:   3692988.00
Epoch  11, Loss:   6963352.00
Epoch  12, Loss:   2915028.00
Epoch  13, Loss:   1103360.00
Epoch  14, Loss:   1270616.00
Epoch  15, Loss:   1196528.00
Epoch  16, Loss:    708336.00
Epoch  17, Loss:    763440.00
Epoch  18, Loss:    662072.00
Epoch  19, Loss:    729496.00
Epoch   0, Loss:   6303330.50
Epoch   1, Loss:   4213216.50
Epoch   2, Loss:   4078188.00
Epoch   3, Loss:   3760537.00
Epoch   4, Loss:   3049088.00
Epoch   5, Loss:   4568900.00
Epoch   6, Loss:   4542260.00
Epoch   7, Loss:   6918476.00
Epoch   8, Loss:   5319080.00
Epoch   9, Loss:   2778784.00
Epoch  10, Loss:   6760024.00
Epoch  11, Loss:   3738264.00
Epoch  12, Loss:   5377604.00
Epoch  13, Loss:   4258044.00
Epoch  14, Loss:   1972420.00
Epoch  15, Loss:   1289976.00
Epoch  16, Loss:   1211640.00
Epoch  17, Loss:   1040168.00
Epoch  18, Loss:    549352.00
Epoch  19, Loss:    611424.00
```

This function is used to get the dimension vector for a concept (e.g., gender, race or class). I've mostly used Kozlowski's proposed pairs (as long as the vocabulary item was defined) and I've also added some of my own.

In [12]:

```
def get_dimension_vector(concept_name, w2v):
    """
    Returns the dimension vector (direction of dimension)
    of a certain concept.
    :param concept_name: the concept to get the dimension vector for
    :param w2v: the word embedding
    :return: the vector pointing into the direction of the concept
    """

    # determine pairs to compute dimension direction
    pairs = None
    if concept_name == 'gender':
        pairs = [
            ('man', 'woman'),
            ('men', 'women'),
            # ('he', 'she'),
            # ('him', 'her'),
            # ('his', 'hers'),
            ('boy', 'girl'),
            ('boys', 'girls'),
            ('male', 'female'),
            # ('masculine', 'feminine')
        ]
    elif concept_name == 'race':
        pairs = [
            ('black', 'white'),
            ('blacks', 'whites'),
            ('african', 'european'),
            # ('african', 'caucasian'),
        ]
    elif concept_name == 'class':
        pairs = [
            ('rich', 'poor'),
            #('richer', 'poorer'),
            #('richest', 'poorest'),
            #('affluence', 'poverty'),
            #('affluent', 'impoverished'),
            #- ('expensive', 'inexpensive'),
            #- ('luxury', 'cheap'),
            #('opulent', 'needy')
            # ----------------------
            ('rich', 'broke'),
            ('wealthy', 'poor'),
            ('wealthy', 'broke'),
            #- ('expensive', 'cheap'),
            #- ('costly', 'economical'),
            #- ('precious', 'worthless'),
            #('luxurious', 'spartan'),
            #('aristocratic', 'lower-class'),
            #('aristocratic', 'plebeian'),
            #('upper', 'lower'), # class
        ]
    else:
        raise RuntimeError('invalid concept_name!')

    # compute dimension direction through average
    avg = np.zeros(w2v.vector_size)
    for pair in pairs:
        current = w2v.wv[pair[0]] - w2v.wv[pair[1]]
```

```
        avg += current
    avg /= np.linalg.norm(avg)
    return avg
```

This function is used to compute the x/y-coordinates of terms in a topic according to two concepts.

In [13]:

```
def compute_positions(w2v, concepts, topic):
    """
    Computes the position of terms in a topic according to two concepts
    in a word embedding
    :param w2v: the word embedding to use
    :param concepts: the concepts used as x and y axis
    :param topic: the topics to compute the positions for according to the
    concepts
    :return: positions (x,y) of the terms in the topic according to concepts
    """

    # create a data frame containing a row for each topic
    positions = pd.DataFrame(list(enumerate(topic, 0)),
                             columns=['word_index', 'word'])

    # get concept dimension directions
    concept_1_dim = get_dimension_vector(concepts[0], w2v)
    concept_2_dim = get_dimension_vector(concepts[1], w2v)

    # initialize x and y coordinates for terms in topic
    x = np.zeros(len(topic))
    y = np.zeros(len(topic))

    # for each topic in the topic
    for i, term in enumerate(topic, 0):
        # get the embedding of that instance
        term_embedding = w2v.wv[term]
        # determine association to concepts
        # determine (x/y)-coordinates
        x[i] = np.dot(concept_1_dim, term_embedding)
        y[i] = np.dot(concept_2_dim, term_embedding)

    # assign x and y coordinates
    positions['x'] = x
    positions['y'] = y

    return positions
```

This function shows the comparision of positions according to two concepts of terms in a topic for republicans and democrats.

In [24]:

```
def show_comparison_chart(rep_data, dem_data, concepts, topic):

    # initialize plot
    plt.figure(figsize=(16,16))

    # extract coordinates
    x_coords1 = rep_data['x']
    y_coords1 = rep_data['y']
    x_coords2 = dem_data['x']
    y_coords2 = dem_data['y']

    # determine axis span
    ax_max = np.max(np.abs(np.concatenate((x_coords1, x_coords2, y_coords1, y_co
ords2))))*1.2

    # keep track of last point for legend
    rep = None
    dem = None

    # define text offset
    txt_off = 0.003

    # for each term in the topic
    for i, term in enumerate(topic):
        # get the coordinates of the first point
        x1 = x_coords1[i]
        y1 = y_coords1[i]
        # get the coordinates of the second point
        x2 = x_coords2[i]
        y2 = y_coords2[i]
        # create first point (republican)
        rep = plt.scatter(x1, y1, marker='.', color='red')
        plt.text(x1 + txt_off, y1 + txt_off, term, fontsize=22, color='red')
        # create second point (democrat)
        dem = plt.scatter(x2, y2, marker='.', color='blue')
        plt.text(x2 + txt_off, y2 + txt_off, term, fontsize=22, color='blue')
        # connect points with line
        plt.plot([x1, x2], [y1, y2], color='gray', alpha=0.3)

    # add legend
    plt.legend([rep, dem], ['Republican', 'Democrat'], loc='upper right')

    # adjust axes
    axes = plt.gca()
    axes.set_xlim([-ax_max, ax_max])
    axes.set_ylim([-ax_max, ax_max])
    axes.set_aspect('equal', 'box')
    ylim = axes.get_ylim()
    xlim = axes.get_xlim()
    plt.vlines(0, ylim[0], ylim[1], colors='gray')
    plt.hlines(0, xlim[0], xlim[1], colors='gray')

    # define concept to axis title mapping
    concept_to_titles = {
        'gender': ['female', 'Gender', 'male'],
        'race': ['white', 'Race', 'black'],
        'class': ['poor', 'Class', 'rich']
    }
```

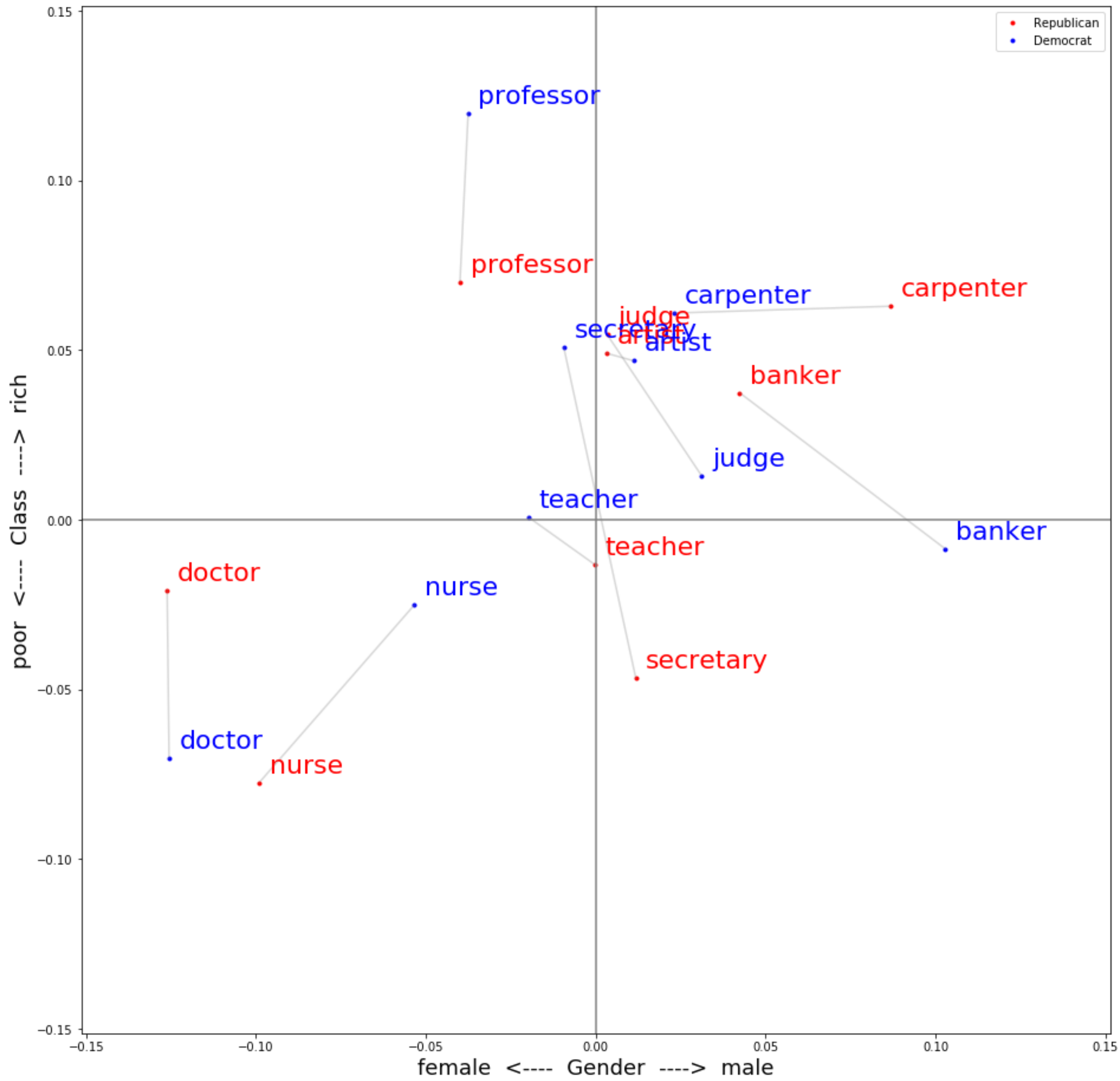
```
    # get axis names
    x_lo = concept_to_titles[concepts[0]][0]
    x_mi = concept_to_titles[concepts[0]][1]
    x_hi = concept_to_titles[concepts[0]][2]
    y_lo = concept_to_titles[concepts[1]][0]
    y_mi = concept_to_titles[concepts[1]][1]
    y_hi = concept_to_titles[concepts[1]][2]

    # annotate axes with axis names
    plt.xlabel(x_lo + '  <----  '+x_mi+'  ---->  '+x_hi, fontsize=18)
    plt.ylabel(y_lo + '  <----  '+y_mi+'  ---->  '+y_hi, fontsize=18)

    plt.show()
```

Now let's analyze how certain terms of a topic are placed (according to two concepts) in the word embeddings for democrats and republicans

In our first experiment we'll analyze how jobs are positioned according to gender and class:

In [25]:

```
# create job topic
jobs = [
    'nurse',
    'carpenter',
    'judge',
    'secretary',
    'banker',
    'artist',
    'doctor',
    'teacher',
    'professor'
]
# use gender and class concepts
concepts = ['gender', 'class']
# get position of topic items according to concepts for each party
rep_data = compute_positions(w2v_republican, concepts, jobs)
dem_data = compute_positions(w2v_democrat, concepts, jobs)
# show the comparison plot
show_comparison_chart(rep_data, dem_data, concepts, jobs)
```

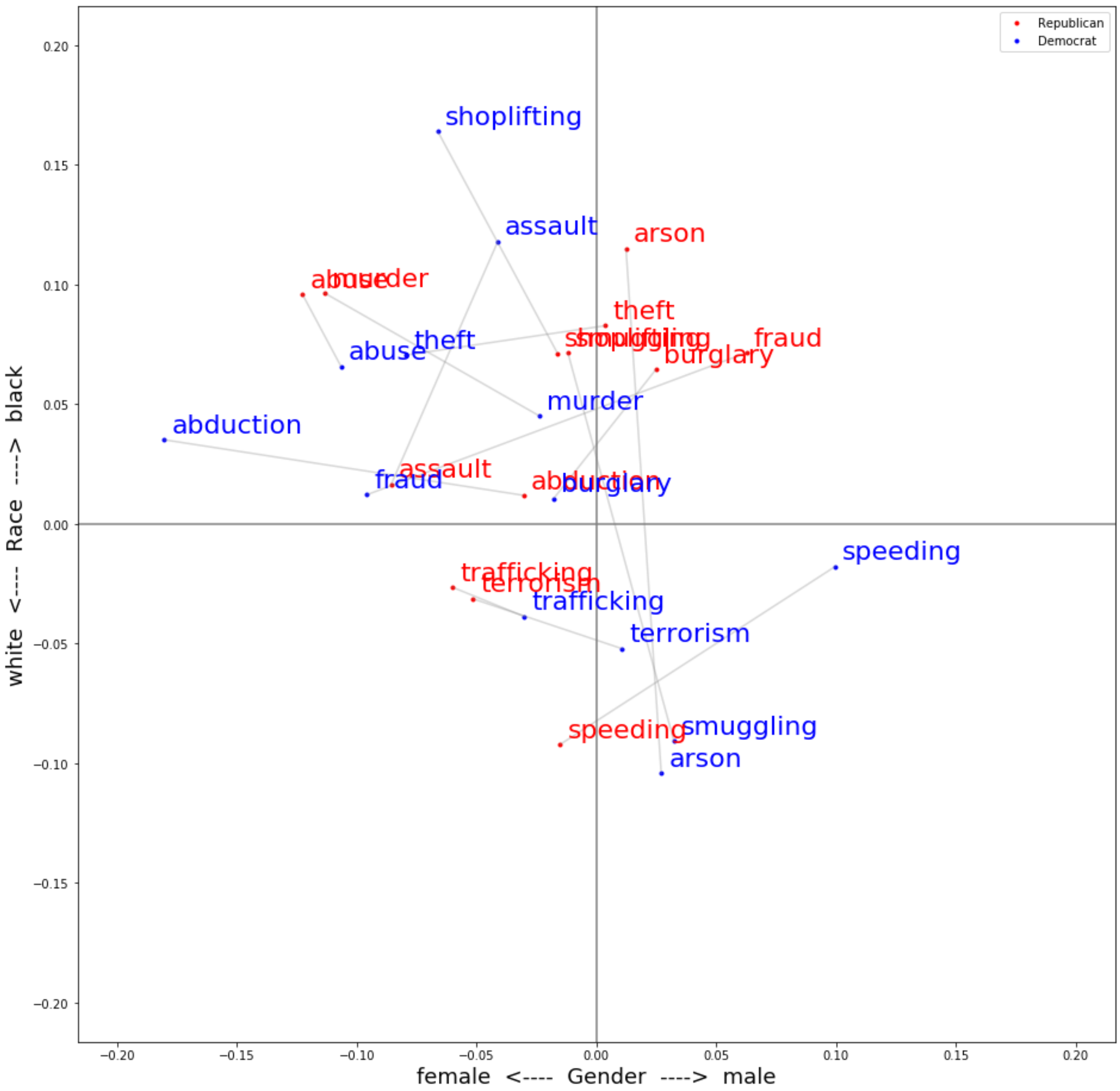


As one can see jobs like 'carpenter' and 'banker' are are associated with 'male' and 'nurse' is associated with 'female'. There aren't any significant differences between the republican and democrat embeddings. A surprising thing is that 'doctor' is the term most associated with 'female' (even more than nurse) in both embeddings. Another surprising fact is that 'professor' is the job that is the most associated with 'rich' - even more than the job 'banker'. The terms 'carpenter' and 'professore' are associated more with prosperity than the term 'banker'.

In our second experiment we'll analyze how crimes are positioned according to gender and race:

In [26]:

```python
# create crime topic
crimes = [
    'murder',
    'assault',
    'theft',
    'trafficking',
    'terrorism',
    'smuggling',
    'shoplifting',
    'speeding',
    'abduction',
    'arson',
    'burglary',
    'abuse',
    'fraud'
]
# use gender and race concepts
concepts = ['gender', 'race']
# get position of topic items according to concepts for each party
rep_data = compute_positions(w2v_republican, concepts, crimes)
dem_data = compute_positions(w2v_democrat, concepts, crimes)
# show the comparison plot
show_comparison_chart(rep_data, dem_data, concepts, crimes)
```


Republican
Democrat
shoplifting
assault
arson
abuse
murder
theft
abuse
theft
smuggling
shoplifting
fraud
burglary
murder
abduction
assault
fraud
abduction
burglary
speeding
trafficking
terrorism
trafficking
terrorism
speeding
smuggling
arson
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
white <---- Race ----> black
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
female <---- Gender ----> male

As one can see, the crimes 'abduction', 'abuse', 'shoplifting' and assault are all associadet with the 'black'-'female' quadrant for both embeddings. The crimes 'speeding', 'trafficking' and 'terrorism' are all in the 'white' half-plane for both embeddings. The crimes 'abduction', 'fraud', 'assault', 'murder', 'theft', 'abuse', 'shoplifting', 'burglary' are all in the 'black' half-plane for both embeddings. An interesting fact is that 'arson' and 'smuggling' are in completely different 'race'-halfplanes in each of the embeddings. A crime that associates strongly with 'female' in both embeddings is 'abuse'. The republican embedding places no crime in the 'white'-'male' quadrant, whereas the democrat embedding places no crime in the 'black'-'male' quadrant.

Exercise 3

Q: Implement one of the causal inference methods from Lecture 12 (choose one of the following options):

- Look at the effect of text features on citations or reversal (Fong and Grimmer 2016, Hartford et al 2017, or Wang and Blei 2018)
- Look at the effect of political party on citations or reversal holding text features constant (Roberts-Stewart-Nielsen 2018 matching method or Chernozhukov et al 2017 double ML method).
- Look at heterogeneous effects of political party on citations or reversal depending on text features (Wager and Athey 2017).

A: I'm using the double ML method (Chernozhukov et al 2017) to look at the effect of political party on reversal. So, we have:

- T: the political party of the judge (binary variable)
- A: the constant document features (high dim. confounders)
- Y: the reversal decision of the case (binary variable) What we ant to find is the effect θ of the relationship:
  $$Y = \theta T + g(A) + \epsilon,$$
  where
  $$T = m(A) + \eta.$$
  Helpful resources I've used to build this code were:
- https://arxiv.org/pdf/1608.00060.pdf (https://arxiv.org/pdf/1608.00060.pdf)
- https://gist.github.com/sergeyf/4bb4b3141fc0ac583ddfa929a9da19f8 (https://gist.github.com/sergeyf/4bb4b3141fc0ac583ddfa929a9da19f8)
- https://medium.com/teconomics-blog/using-ml-to-resolve-experiments-faster-bd8053ff602e (https://medium.com/teconomics-blog/using-ml-to-resolve-experiments-faster-bd8053ff602e)
- https://insightr.wordpress.com/2017/06/28/cross-fitting-double-machine-learning-estimator/ (https://insightr.wordpress.com/2017/06/28/cross-fitting-double-machine-learning-estimator/)

First we define a function to get the dataset for exercise 3

In [72]:

```
# parameters
N_GRAM_LENGTH = 3
NUM_FEATURES_MOST_COMMON = 10000
CASE_METADATA = 'case_metadata.csv'

def get_exercise_3_dataset():

    # create message counter
    counter = CounterMessage()

    # open cases zip file
    zfile = ZipFile('data/cases.zip')
    caseids = []
    raw_texts = {}
    years = {}

    # randomly shuffle files
    members = zfile.namelist()
    NUM_CASES = len(members)

    for case in members:
        year, caseid = case[:-4].split('_')
        with zfile.open(case) as f:
            raw_text = f.read().decode()
        raw_texts[caseid] = raw_text
        years[caseid] = int(year)
        caseids.append(caseid)
        counter.update(NUM_CASES, 'opened')

    # do NLP
    nlp = spacy.load('en')
    spacy_documents = {}
    for caseid in caseids:
        spacy_documents[caseid] = nlp(raw_texts[caseid])
        counter.update(NUM_CASES, 'nlp-processed')

    # create punctuation remover
    punctuation_remover = str.maketrans('', '', punctuation)
    # create lemmatizer
    lemmatizer = Lemmatizer(LEMMA_INDEX, LEMMA_EXC, LEMMA_RULES)


    def filter_and_transform(token):
        # get the token's word(s)
        word = ''.join(token.text)
        # replace newlines with spaces
        word = word.replace('\r', ' ').replace('\n', ' ')
        # remove punctuation
        word = word.translate(punctuation_remover)
        # replace multiple subsequent spaces with one space
        word = re.sub(' +', ' ', word)
        # check that word still has some text (not just one char or space)
        if len(word) <= 1:
            return False, (word, token.pos_)
        # normalize numbers (28, 28th, 1st, ...)
        if any(char.isdigit() for char in word):
            return False, (word, token.pos_)
        # lemmatize the word
        lemmas = lemmatizer(word, token.pos)[0]
```

```python
        # try to lemmatize the word
        if isinstance(lemmas, (list,)) and len(lemmas) > 0:
            # pick the first option if several lemmas were found
            word = lemmas[0]
        else:
            # no lemma was found (just keep the original word)
            word = word
        # convert the word to lowercase
        word = word.lower()
        # remove stopwords
        if word in STOP_WORDS:
            return False, (word, token.pos_)
        # finally, return the filtered word and type
        return True, (word, token.pos_)

    case_ngrams = {}
    all_ngrams = []
    all_case_tokens = []

    stemmer = SnowballStemmer('english')

    # n-gram cases
    for caseid in caseids:
        spacy_document = spacy_documents[caseid]
        # process each sentence separately
        # (we don't want n-grams to overlap sentences)
        case_tokens = []
        case_noun_ngrams = []
        for sentence in spacy_document.sents:
            sentence_tokens = []
            # filter the tokens in the case
            for token in sentence:
                take_token, filtered_token = filter_and_transform(token)
                if take_token:
                    sentence_tokens.append(filtered_token)
            # append list of processed tokens for this document
            tl = [stemmer.stem(t[0]) for t in sentence_tokens]
            case_tokens += (tl)
            # create list to keep track of all n-grams ending in a noun
            # in this sentence
            case_sentence_noun_ngrams = []
            # iterate over all ngrams that can be built out of the tokens
            for ngram in ngrams(sentence_tokens, N_GRAM_LENGTH):
                # check if the last word is a noun
                if ngram[N_GRAM_LENGTH-1][1] == 'NOUN':
                    # if so, add that n-gram
                    curr_ngram = (ngram[0][0],
                                  ngram[1][0],
                                  ngram[2][0])
                    # stores all n-grams for this sentence of this case
                    case_sentence_noun_ngrams.append(curr_ngram)
                    # stores all n-grams for all cases
                    all_ngrams.append(curr_ngram)
            # save all n-grams appearing in sentence
            case_noun_ngrams += case_sentence_noun_ngrams
        # save list of all n-grams for this case
        case_ngrams[caseid] = case_noun_ngrams
        counter.update(NUM_CASES, 'n-grammed')
        # save list of all appearing tokens for this case
        all_case_tokens.append(case_tokens)
```

```
    # load metadata into dictionary
    metadata = {}
    case_metadata = pd.read_csv(os.path.join(DATA_DIR, CASE_METADATA)).values
    for caseid, case_reversed, judge_id, year, republican, log_cites in case_met
adata:
        if not (isnan(republican) or isnan(case_reversed)):
            metadata[caseid] = {
                'reversed': case_reversed,
                'judge_id': judge_id,
                'year_meta': year,
                'republican': republican,
                'log_cites': log_cites
            }

    # create list of values
    T = []
    A = []
    Y = []

    # determine most common n_grams
    most_common = Counter(all_ngrams).most_common(NUM_FEATURES_MOST_COMMON)

    # featurize according to most common n_grams
    for caseid in caseids:
        if caseid in metadata:
            # count the n_gram frequencies of the current case
            current_case_ngrams = Counter(case_ngrams[caseid])
            # create feature vector
            features = np.zeros(len(most_common))
            # for each most common ngram
            for i in range(len(most_common)):
                ngram = most_common[i][0]
                # check if it appears in the cases's n_grams
                if ngram in current_case_ngrams:
                    # if it appears, add the number of appearances as a feature
                    features[i] = current_case_ngrams[ngram]
            # create feature vectors and targets for reversed
            T.append(metadata[caseid]['republican'])
            A.append(features)
            Y.append(metadata[caseid]['reversed'])
        counter.update(NUM_CASES, 'featurized')

    # convert to numpy array
    T = np.array(T)
    A = np.array(A)
    Y = np.array(Y)

    # standardize maintain sparsity by not subtracting the mean
    A = A / np.std(A, axis=0)  # standardize features preserving sparsity

    # remove zero variance columns which gave NaN
    A = A.transpose()
    A = A[~np.isnan(A).any(axis=1)]
    A = A.transpose()

    # return dataset
    return T, A, Y
```

Now we'll load the dataset

In [73]:

```
T, A, Y = get_exercise_3_dataset()
```

```
5762/5762: opened.
5762/5762: nlp-processed.
5762/5762: n-grammed.
5762/5762: featurized.
```

```
/local/home/abloch/venv/lib/python3.5/site-packages/ipykernel_launch
er.py:168: RuntimeWarning: invalid value encountered in true_divide
```

(The division by zero is no problem, as I'm removing the columns with zero variance)

And now we'll estimate theta using the double ML method:

In [79]:

```python
def get_theta_est(A_train, Y_train, T_train,
                  A_test, Y_test, T_test,
                  regressor, classifier):
    """
    Estimates the effect theta using the Double ML method
    :param A_train: training confounders A (high dim.)
    :param Y_train: training outcome
    :param T_train: training treatment
    :param A_test: test confounders A (high dim.)
    :param Y_test: test outcome
    :param T_test: test treatment
    :param regressor: A -> T
    :param classifier: A -> Y
    :return:
    """

    # estimate T and Y based on any ML method
    T_test_pred = classifier.fit(A_train, T_train).predict_proba(A_test)[:, 1]
    Y_test_pred = regressor.fit(A_train, Y_train).predict(A_test)

    # compute residuals (equation 1.5 in paper)
    T_residual = T_test - T_test_pred
    Y_residual = Y_test - Y_test_pred

    # get estimate of theta (by regressing Y on t)
    theta_est = np.mean(T_residual * Y_residual) / np.mean(T_residual * T_test)

    return theta_est


# create regressor and classifier
regressor = RandomForestRegressor(n_estimators=10)
classifier = RandomForestClassifier(n_estimators=10)

# keep track of estimators
theta_ests = []

# do sample-splitting to remove bias induced by overfitting
splitter = KFold(n_splits=10, shuffle=True)
for tr, ts in tqdm(splitter.split(A)):
    theta_est = get_theta_est(A[tr, :], Y[tr], T[tr],
                              A[ts, :], Y[ts], T[ts],
                              regressor, classifier)
    theta_ests.append(theta_est)

# compute final theta estimate
theta_est_dml = np.mean(theta_ests)

print('Double ML estimate (using Random Forest) of treatment effect theta:')
print('theta = '+str(theta_est_dml))
```

```
0it [00:00, ?it/s]
1it [13:51, 831.89s/it]
2it [26:50, 815.83s/it]
3it [40:12, 811.64s/it]
4it [53:59, 816.34s/it]
5it [1:08:10, 826.89s/it]
6it [1:21:57, 826.85s/it]
7it [1:34:58, 813.16s/it]
8it [1:47:19, 791.51s/it]
9it [2:02:37, 829.30s/it]
10it [2:17:55, 855.88s/it]

Double ML estimate (using Random Forest) of treatment effect theta:
theta = 0.005909812544524185
```

In [ ]: