

JavaScript Scripts for Capturing and Python Scripts for Processing Client-Based Paradata in Web Surveys

Authors: Nejc Berzelak, Peter Hrvatin, and Vasja Vehovar

*Centre for Social Informatics, Faculty of Social Sciences, University of Ljubljana, Kardeljeva
ploščad 5, 1000 Ljubljana, Slovenia*

Funding:

This work was supported by the Slovenian Research Agency [grant numbers P5-0399, J5-9334, J5-8233, NI-0004, J5-3100, and V5-2157].

Published by:

Faculty of Social Sciences, Kardeljeva ploščad 5, 1000 Ljubljana, Slovenia

Ljubljana, 2022

1 JavaScript Scripts for Capturing Paradata

1.1 Advanced paradata

```
// The main function that is always called for logging
function logEvent(event_type, event, data){

    //console.log(event_type + ' ' + event + ' ' + _session_id);

    // We get the session id on the page
    var session_id = _session_id;

    // We get the survey id
    var anketa = $('#srv_meta_anketa_id').val();

    // We get the current page
    if (typeof srv_meta_grupa_id != 'undefined'){
        var page = srv_meta_grupa_id;
    }
    else if ($('#outercontainer').hasClass('intro')){
        var page = '-1';
    }
    else if ($('#outercontainer').hasClass('concl')){
        var page = '-2';
    }
    else{
        var page = '0';
    }

    // We get user id, recnum and language
    var user = _usr_id;
    var recnum = _recnum;
    var language = _lang;

    // We note the time of the event
    var timestamp = Date.now();

    // Optional parameter data
    data = data || '';

    // In the submit or unload event, the call must be synchronous
    if(event == 'unload_page'){
        $.ajax({
            type: 'POST',
            url: '../main/survey/ajax.php?t=parapodatki&a=logData',
            data: {
                session_id: session_id,
                anketa: anketa,
                page: page,
                usr_id: user,
                recnum: recnum,
                language: language,

                event_type: event_type,
                event: event,
                timestamp: timestamp,

                data: data
            },
            /*success: success,
            dataType: dataType,*/
            async: false
        });
    }
    else{
        $.post('../main/survey/ajax.php?t=parapodatki&a=logData', {
            session_id: session_id,
            anketa: anketa,
            page: page,
            usr_id: user,
            recnum: recnum,
            language: language,
```

```

        event_type: event_type,
        event: event,
        timestamp: timestamp,

        data: data
    });
}

// Events that we monitor
$(function () {

    // LOAD PAGE
    window.addEventListener('load', function () {

        var event_type = 'page';
        var event = 'load_page';

        // We monitor parameters for size and zoom
        var data = {
            devicePixelRatio: window.devicePixelRatio,
            width: window.screen.width,
            height: window.screen.height,
            availWidth: window.screen.availWidth,
            availHeight: window.screen.availHeight,
            jquery_windowW: $(window).width(),
            jquery_windowH: $(window).height(),
            jquery_documentW: $(document).width(),
            jquery_documentH: $(document).height(),
        }

        // We log load
        logEvent(event_type, event, data);

        // We log visible questions
        //visibleQuestions();
    });

    // RESIZE
    window.addEventListener('resize', function () {

        var data = {
            pos_x: $(window).width(),
            pos_y: $(window).height(),
            value: window.devicePixelRatio
        }

        var event_type = 'other';
        var event = 'resize';

        // We log scroll
        logEvent(event_type, event, data);

        // We log visible questions
        //visibleQuestions();
    });

    // PINCH ZOOM
    var pinch_scaling = false;
    var pinch_distance = 0;
    window.addEventListener('touchstart', function (e) {

        // We detect two fingers
        if (e.touches.length === 2) {
            pinch_scaling = true;

            // We calculate the distance between fingers
            pinch_distance = Math.hypot(
                e.touches[0].screenX - e.touches[1].screenX,
                e.touches[0].screenY - e.touches[1].screenY
            );
        }
    });
    window.addEventListener('touchend', function (e) {

```

```

    if(pinch_scaling){
        pinch_scaling = false;

        // Izracunamo razdaljo med prstoma
        var pinch_distance2 = Math.hypot(
            e.touches[0].screenX - e.changedTouches[0].screenX,
            e.touches[0].screenY - e.changedTouches[0].screenY
        );

        // We calculate the difference in distance
        var diff = pinch_distance2 - pinch_distance;
        var zoom = Math.round((diff / pinch_distance * 100));

        var data = {
            value: zoom
        }

        var event_type = 'other';
        var event = 'pinch_resize';

        // We log scroll
        logEvent(event_type, event, data);
    }
});

// ORIENTATION CHANGE
window.addEventListener('orientationchange', function () {

    var value = '';

    if(window.orientation == 90 || window.orientation == -90)
        value = 'landscape';
    else
        value = 'portrait';

        var data = {
            value: value
        }

        var event_type = 'other';
        var event = 'orientation_change';

        // We log scroll
        logEvent(event_type, event, data);
});

// SCROLL
var scroll_time_prev = 0;
window.addEventListener('scroll', function () {

    // We get the current time
    var scroll_time = new Date().getTime();

    // If enough time has passed since the previous detection
    if ((scroll_time - scroll_time_prev > 50) || scroll_time_prev == 0) {

        var event_type = 'other';
        var event = 'scroll_page';

        var data = {
            pos_x: (window.pageXOffset || document.documentElement.scrollLeft) -
(document.documentElement.clientLeft || 0),
            pos_y: (window.pageYOffset || document.documentElement.scrollTop) -
(document.documentElement.clientTop || 0)
        }

        // We log scroll
        logEvent(event_type, event, data);

        // We log visible questions
        //visibleQuestions();

        // We save the time for the interval
        scroll_time_prev = scroll_time;
    }
});

```

```

    });

    // BLUR (leave to another tab etc..)
    window.addEventListener('blur', function () {

        var event_type = 'other';
        var event = 'blur';

        logEvent(event_type, event);

    });

    // FOCUS
    window.addEventListener('focus', function () {

        var event_type = 'other';
        var event = 'focus';

        logEvent(event_type, event);

    });

    // MOUSE CLICK
    window.addEventListener('click', function (event) {

        var div_id = $(event.target).attr('id');

        // If the id is empty, we find the first parent with the id
        if(div_id == null){
            div_id = $(event.target).closest('[id]').attr('id') + ' (parent)';
        }

        var data = {
            pos_x: event.pageX,
            pos_y: event.pageY,
            div_type: event.target.tagName,
            div_id: div_id,
            div_class: $(event.target).attr('class')
        }

        var event_type = 'other';
        var event_name = 'click';

        logEvent(event_type, event_name, data);

        // We specifically save the mouse movement until this click
        movementClickEvent(event);

    });

    // INPUT TEXT, TEXTAREA FOCUS
    $('input[type=text], textarea').bind('focus', function () {

        // If it is a textbox in the select, it is ignored
        if(!$(this).parent().hasClass('chzn-search')){

            var event_type = 'vrednost';
            var event = 'text_enter';

            var id = $(this).attr('id');
            var id_array = id.split('_');
            var spr_id = id_array[1];
            var vre_id = id_array[3];

            var data = {
                spr_id: spr_id,
                vre_id: vre_id,
                value: $(this).val()
            }

            logEvent(event_type, event, data);

            // Let's check if it's another field - then mark the checkbox/radio as well
            if($(this).parent().parent().parent().hasClass('tip_1') ||
            $(this).parent().parent().parent().hasClass('tip_2')){

```

```

        event = 'radio_checkbox_change';

        data = {
            spr_id: spr_id,
            vre_id: vre_id,
            value: 1
        }

        logEvent(event_type, event, data);
    }
}

});
// INPUT TEXT, TEXTAREA BLUR
$('input[type=text], textarea').bind('blur', function () {

    // If it is a textbox in the select, it is ignored
    if(!$(this).parent().hasClass('chzn-search')){

        var event_type = 'vrednost';
        var event = 'text_leave';

        var id = $(this).attr('id');
        var id_array = id.split('_');
        var spr_id = id_array[1];
        var vre_id = id_array[3];

        var data = {
            spr_id: spr_id,
            vre_id: vre_id,
            value: $(this).val()
        }

        logEvent(event_type, event, data);
    }

});

// INPUT RADIO, CHECKBOX CHANGE
$('input[type=radio], input[type=checkbox]').bind('click', function () {

    var event_type = 'vrednost';
    var event = 'radio_checkbox_change';

    var id = $(this).attr('id');
    var id_array = id.split('_');

    // If it's a table
    if(id_array[2] == 'grid' || id_array[0] == 'grid'){
        var spr_id = id_array[1];
        var vre_id = id_array[3];

        // If it is 'value', it is missing
        if(spr_id == 'missing'){
            spr_id = id_array[3];
            vre_id = id_array[5];
        }
    }
    // Ordinary radio or checkbox
    else{
        var spr_id = id_array[1];
        var vre_id = id_array[3];

        // If it is 'value', it is missing
        if(spr_id == 'value'){
            spr_id = id_array[3];
            vre_id = id_array[5];
        }
    }

    var val;
    if ($(this).is(':checked'))
        val = 1;
    else
        val = 0

    var data = {

```

```

        spr_id: spr_id,
        vre_id: vre_id,
        value: val
    }

    logEvent(event_type, event, data);
});

// SELECT CHANGE
$('select').bind('change', function () {

    var event_type = 'vrednost';
    var event = 'select_change';

    var id = $(this).attr('id');
    var id_array = id.split('_');
    var spr_id = id_array[1];
    var vre_id = $(this).val();

    var data = {
        spr_id: spr_id,
        vre_id: vre_id,
        value: vre_id
    }

    logEvent(event_type, event, data);
});

// MOUSE MOVE
var movements = [];
var currentMovement = {
    timeStart: 0,
    timeEnd: 0,
    duration: 0,

    startPosX: 0,
    startPosY: 0,
    endPosX: 0,
    endPosY: 0,

    distance_traveled: 0,
    prevPosX: 0,
    prevPosY: 0
}

// We capture all mouse movements
window.addEventListener('mousemove', function (event) {
    var clicked = false;
    movementTrack(event, clicked);
});

// We record the movement in the array of objects
var time_prev = new Date().getTime();
function movementTrack(event, clicked){

    // We get the current time
    var time = new Date().getTime();

    // If enough time has passed since the previous detection
    if (time - time_prev > 50 || clicked) {

        // Let's check if this is already a new movement - for now we treat a new movement
        if the pause is more than 300ms
        if(time - currentMovement.timeEnd > 300 || clicked){

            // We add the current movement to the movements array - the trick is not to
            insert a reference but an actual copy of the object
            if(currentMovement.timeStart !== 0 && currentMovement.startPosX !== undefined
            && currentMovement.endPosX !== undefined){
                movements.push(JSON.parse(JSON.stringify(currentMovement)));
            }

            // We reset the current movement and set all its parameters
            currentMovement.timeStart = time;

```

```

        currentMovement.timeEnd = time;

        currentMovement.startPosX = event.pageX;
        currentMovement.startPosY = event.pageY;

        currentMovement.endPosX = event.pageX;
        currentMovement.endPosY = event.pageY;

        currentMovement.distance_traveled = 0;
        currentMovement.prevPosX = event.pageX;
        currentMovement.prevPosY = event.pageY;
    }
    // It's an old move - we just update the end time and end position
    else{
        currentMovement.timeEnd = time;
        currentMovement.duration = time - currentMovement.timeStart;

        currentMovement.endPosX = event.pageX;
        currentMovement.endPosY = event.pageY;

        // We calculate the distance traveled since the previous move
        var a = currentMovement.prevPosX - event.pageX;
        var b = currentMovement.prevPosY - event.pageY;
        var distance = Math.sqrt(a*a + b*b);

        currentMovement.distance_traveled += distance;
        currentMovement.prevPosX = event.pageX;
        currentMovement.prevPosY = event.pageY;
    }

    // Let's save the time for the interval
    time_prev = time;

    /*console.log(currentMovement);*/
}

// We call on every click and save data about the movement during the click
function movementClickEvent(event) {

    var event_type = 'movement';
    var event = 'mouse_move';

    var clicked = true;
    movementTrack(event, clicked);

    // Loop through all moves
    for (var i=0; i<movements.length; i++) {

        var data = {
            time_start: movements[i].timeStart,
            time_end: movements[i].timeEnd,
            pos_x_start: movements[i].startPosX,
            pos_y_start: movements[i].startPosY,
            pos_x_end: movements[i].endPosX,
            pos_y_end: movements[i].endPosY,
            distance: movements[i].distance_traveled
        }

        logEvent(event_type, event, data);
    }

    // Let's clear the array of all movements
    movements = [];
}

// VISIBLE
/*function visibleQuestions() {

    var event_type = 'other';
    var event = 'visible_question';

    var q = $('.grupa').find('.spremenljivka');
    var arr = [];
}

```



```

$.each(q, function (i, val) {
    if ($(val).is(':visible') && isVisible(val)) {
        if ($(val).attr('id'))
            arr[arr.length] = $(val).attr('id').substring(14);
    }
})

var log = "" + arr;

if (log != prev_log) {
    var data = {
        log: log
    }

    logEvent(event_type, event, data);

    prev_log = log;
}
var prev_log = '';

// We return if the element is visible
function isVisible(elem) {

    var containerTop = $(window).scrollTop();
    var containerBottom = containerTop + $(window).height();

    var elemTop = $(elem).offset().top;
    var elemBottom = elemTop + $(elem).height();

    return ((elemBottom >= containerTop) && (elemTop <= containerBottom)
    && (elemBottom <= containerBottom) && (elemTop >= containerTop) );
}*/
})

```

1.2 Post time

```

// EVENT LEAVE PAGE, which must be executed synchronously (otherwise it will be lost in some
browsers)
$(function () {

    // LEAVE PAGE (we look at unload, beforeunload, click on submit, click on back)
    var leavePageFunction_called = false;
    var leavePageFunction = function () {
        var event_type = 'page';
        var event = 'unload_page';

        if(!leavePageFunction_called){
            logEvent(event_type, event);
            leavePageFunction_called = true;
        }
    }
    window.addEventListener('beforeunload', leavePageFunction);
    window.addEventListener('unload', leavePageFunction);
    $("input.next:submit").bind('click', leavePageFunction);
    $("input.prev:button").bind('click', leavePageFunction);

})

```

1.3 Prompts

```

var spr_id_variable = []; // to track alerts: a field that holds the
spr_id where alerts occur
var tip_opozorila = []; // for alert tracking: a field that holds
the reminder/alert type
var spr_id indeks = 0; // to track alerts: index to the
fields that hold the spr_id where the alerts occur and the reminder type
var opozorila_sum = []; // to track alerts
var opozorila_num = []; // to track alerts
var opozorila_validation = []; // to track alerts

```

```

var tip_opozorila_temp = [];
var validacijaZabelezena = [];
var zacetnaValidacijaZabelezena = 1; // records if the validation was recorded, not
dynamically, without the respondent clicking on any answer and going directly to the next page
of the survey
var spremenljivkaVal = []; // a field that records which variables
have a wave warning

// WE DETECT THE TRIGGER OF THE ALERT
$(function () {

    var time_display;

    // remember the normal alert
    var oldAlert = (function(){ return this.alert; }()),
    oldConfirm = (function(){ return this.confirm; }());

    // inject ourself into the window.alert and window.confirm globals
    alert = function (msg) {
        time_display = Date.now();

        oldAlert.call(window, msg);
        window.onAlert(msg);
    };
    confirm = function (msg) {
        time_display = Date.now();

        var result = oldConfirm.call(window, msg);
        window.onConfirm(msg, result);

        return result;
    };

    // these just listen for events
    window.onAlert = function (text) {
        logAlert({text:text, type:'alert box', ignorable:0, action:'ok',
time_display:time_display});
    };
    window.onConfirm = function (text, result) {
        var action = result ? 'yes' : 'no';
        logAlert({text:text, type:'confirm box', ignorable:1, action:action,
time_display:time_display});
    };
});

// We log an alert (upon submission)
function logAlert(box_data){

    //console.log("Trenutna dolzina polja: "+spr_id_variable.length);
    //console.log(box_data);

    var spremenljivkaVal = []; // a field that stores the spr_id of variables with a
triggered val warning
    var tip_opozorila_tmp = [];

    spr_id_variable.forEach(function(variable, index) {

        // if the warning type contains a comma, it means that it records two warnings
at the same time
        // break the warning into two parts
        if(tip_opozorila[variable].includes(",")){
            var opozorilo = tip_opozorila[variable].split(",");
            opozorilo[1] = opozorilo[1].substring(1); // remove the space at the
beginning of the warning

            //$.post('../main/survey/ajax.php?t=parapodatki&a=logData', {usr_id:
_usr_id, what: opozorilo[0], what2: opozorilo[1], gru_id: page, anketa: srv_meta_anketa_id,
spr_id_variable: variable});
            var event_type = 'alert';
            var event = 'alert';

```

```

        opozorilo_type[0] = opozorilo[0].substring(4);
        opozorilo_trigger_type[0] = opozorilo[0].substring(0, 3);
        opozorilo_type[1] = opozorilo[1].substring(4);
        opozorilo_trigger_type[1] = opozorilo[1].substring(0, 3);

        var data = {
            type: opozorilo_type[0] + ' ' + opozorilo_type[1] + ' (' +
box_data.type + ')',
            trigger_id: variable,
            trigger_type: opozorilo_trigger_type[0] + ' ' +
opozorilo_trigger_type[1],
            text: box_data.text,
            ignorable: box_data.ignorable,
            action: box_data.action,
            time_display: box_data.time_display
        };

        logEvent(event_type, event, data);
    }
    else{
        //$.post('../main/survey/ajax.php?t=parapodatki&a=logData', {usr_id:
_usr_id, what: tip_opozorila[variable], gru_id: page, anketa: srv_meta_anketa_id,
spr_id_variable: variable});
        var event_type = 'alert';
        var event = 'alert';

        var opozorilo_type = tip_opozorila[variable].substring(4);
        var opozorilo_trigger_type = tip_opozorila[variable].substring(0, 3);

        // If it is a multigrid, we only record the id of the question
        if(variable.substring(0, 12) == '#vrednost_if'){

            var spremenljivka_id =
$(" "+variable).closest('.spremenljivka').attr('id').substring(14);

            //variable = spremenljivka_id + '_' + variable.substring(13);
            variable = spremenljivka_id;
        }

        var data = {
            type: opozorilo_type + ' (' + box_data.type + ')',
            trigger_id: variable,
            trigger_type: opozorilo_trigger_type,
            text: box_data.text,
            ignorable: box_data.ignorable,
            action: box_data.action,
            time_display: box_data.time_display
        };

        logEvent(event_type, event, data);
    }

    //console.log("Spr_id opozorila: "+variable+", indeks "+index+",
tip:"+tip_opozorila[variable]+" za user: "+_usr_id+" zacetnaValidacijaZabelezena:
"+zacetnaValidacijaZabelezena);

    // if no initial validation is recorded and the warning type is validation
    if(zacetnaValidacijaZabelezena == 0 && tip_opozorila[variable].includes("val"))
    {
        spremenljivkaVal.push(variable);

        // if a double warning is recorded
        if(tip_opozorila[variable].includes(",")){
            tip_opozorila_tmp[variable] = opozorilo[0];
        }
        else{
            tip_opozorila_tmp[variable] = tip_opozorila[variable];
        }
    }
}

});

spr_id_variable = []; // clearing the field
tip_opozorila = [];

if(spremenljivkaVal.length != 0){
    spremenljivkaVal.forEach(function(valSprem) {

```

```

        spr_id_variable.push(valSprem);
        tip_opozorila[valSprem] = tip_opozorila_tmp[valSprem];
    });
}

//console.log("Polje spremenljivkaVal: "+spremenljivkaVal.length);
}

function dodaj_opozorilo_val(bol, id){

    var spr_id = id.replace('#spremenljivka_', '');

    //console.log("Tip opozorila prej:" + tip_opozorila[spr_id]);

    if(zacetnaValidacijaZabelezena == 1){
        zacetnaValidacijaZabelezena = 0;
    }
    else{
    }

    if(!validacijaZabelezena[spr_id]){
        spr_id_variable.push(spr_id);
    }

    var tip_alerta = '';

    tip_alerta = 'val';
    validacijaZabelezena[spr_id] = 1;
    spremenljivkaVal[spr_id] = spr_id;

    tip_opozorila[spr_id] = tip_alerta + ' ' + bol + ' alert';
}

function dodaj_opozorilo(alert_sum, alert_num, alert_validation, bol, id){

    //console.log("Dodaj opozorilo");

    var spr_id = id.replace('#spremenljivka_', '');

    var tip_alerta = '';

    if(alert_sum) tip_alerta = 'sum';
    if(alert_num) tip_alerta = 'num';

    if(tip_alerta == ''){

        // if no warning has been recorded for this variable
        if(tip_opozorila[spr_id] == undefined){
            tip_opozorila[spr_id] = bol+' alert';
            spr_id_variable.push(spr_id);
        }
        else{
            // otherwise, if some kind of warning is recorded, usually when there is
validation
            tip_opozorila_temp[spr_id] = bol+' alert';

            // if the previous warning is the same as the current one
            if(tip_opozorila[spr_id] == tip_opozorila_temp[spr_id]){

                tip_opozorila[spr_id] = tip_opozorila_temp[spr_id];
                tip_opozorila_temp[spr_id] = '';

                spr_id_variable.push(spr_id);
            }
            // otherwise, we have two different warnings that need to be noted
            else if(validacijaZabelezena[spr_id] == 1){

                tip_opozorila[spr_id] = tip_opozorila[spr_id]+' '+bol+' alert';
                tip_opozorila_temp[spr_id] = '';

                //console.log("Dvojno opozorilo");
            }
        }
    }
}

```

```

    }
    else{
        // if no warning has been recorded for this variable
        if(tip_opozorila[spr_id] == undefined){

            tip_opozorila[spr_id] = tip_alerta+' '+bol+' alert';
            spr_id_variable.push(spr_id);
        }
        // otherwise, if some kind of warning is recorded, usually when there is
validation
        else{
            tip_opozorila_temp[spr_id] = tip_alerta+' '+bol+' alert';

            // if the previous warning is the same as the current one
            if(tip_opozorila[spr_id] == tip_opozorila_temp[spr_id]){
                tip_opozorila[spr_id] = tip_opozorila_temp[spr_id];
                tip_opozorila_temp[spr_id] = '';
                spr_id_variable.push(spr_id);
            }
            // otherwise, we have two different warnings that need to be noted
            else if(validacijaZabelezena[spr_id] == 1){
                tip_opozorila[spr_id] = tip_opozorila[spr_id]+' '+tip_alerta+'
'+bol+' alert';
                tip_opozorila_temp[spr_id] = '';
            }
        }
    }
}

function odstrani_opozorilo(id, alert_sum, alert_num, alert_validation){
    //console.log("Odstrani");

    var spr_id = id.replace('#spremenljivka_', '');

    tip_opozorila.splice(spr_id, 1);    // remove the warning from the field

    if(alert_validation){
        // remove the recorded validation from the field
        spr_id_variable.forEach(function(variable, index) {
            if(variable == spremljivkaVal[variable]){
                spr_id_variable.splice(index, 1);
            }
        });

        validacijaZabelezena[spr_id] = 0;
    }
}

```

2 Python Scripts for Processing Paradata

2.1 Metadata parser

```
def metadata_parse(file):
    import pandas as pd
    import numpy as np
    import json
    from pandas.io.json import json_normalize

    with open(file) as json_file:
        json_import = json.load(json_file)

    # Create multiple dataframes to store relevant information at different levels.
    Workarounds by parsing individual
    # dataframe columns containing lower-level JSON tags is used due to problems handling non-
    # existant tags.
    # TODO: Make more efficient (e.g. without loops where possible)

    # Parse and edit metadata at the level of pages.
    pages = json_normalize(json_import, record_path=['survey', 'questionnaire', 'pages'],
                           meta=[["survey", "id"]]).copy()
    pages.rename(columns={"id": "page_id", "survey.id": "survey_id"}, inplace=True)

    # Get questions from pages and remove questions metadata from the pages dataframe
    questions_json = pages[pages["questions"].isnull() == False][["survey_id", "page_id",
"questions"]]
    pages.drop(columns="questions", inplace=True)

    # Parse and edit metadata at the level of questions. Higher level identifiers are kept for
    # easier handling.
    questions = pd.DataFrame(columns=["survey_id", "page_id"])
    for i_row in range(0, questions_json.shape[0]):
        i_row_questions = json_normalize(questions_json.iloc[i_row, 2])
        i_row_questions = i_row_questions.assign(survey_id=questions_json.iloc[i_row, 0],
                                                page_id=questions_json.iloc[i_row, 1])
        questions = questions.append(i_row_questions, sort=False, ignore_index=True)
    questions.rename(columns={"id": "question_id"}, inplace=True)
    questions[["survey_id", "page_id", "question_id"]] = questions[["survey_id", "page_id",
"question_id"]].astype(
        np.int64)

    # Get items and response options from questions and remove these columns from the
    # questions dataframe.
    items_json = questions[questions["items"].isnull() == False][
        ["survey_id", "page_id", "question_id", "items"]].copy()
    response_options_json = questions[questions["answers"].isnull() == False][
        ["survey_id", "page_id", "question_id", "answers"]].copy()
    questions.drop(columns=["items", "answers"], inplace=True)

    items = pd.DataFrame(columns=["survey_id", "page_id", "question_id"])
    for i_row in range(0, items_json.shape[0]):
        i_row_items = json_normalize(items_json.iloc[i_row, 3])
        i_row_items = i_row_items.assign(survey_id=items_json.iloc[i_row, 0],
                                        page_id=items_json.iloc[i_row, 1],
                                        question_id=items_json.iloc[i_row, 2])
        items = items.append(i_row_items, sort=False, ignore_index=True)
    items.rename(columns={"id": "item_id"}, inplace=True)
    items[["survey_id", "page_id", "question_id", "item_id"]] = items[
        ["survey_id", "page_id", "question_id", "item_id"]].astype(np.int64)

    response_options = pd.DataFrame(columns=["survey_id", "page_id", "question_id"])
    for i_row in range(0, response_options_json.shape[0]):
        i_row_response_options = json_normalize(response_options_json.iloc[i_row, 3])
        i_row_response_options =
    i_row_response_options.assign(survey_id=response_options_json.iloc[i_row, 0],
    page_id=response_options_json.iloc[i_row, 1],
    question_id=response_options_json.iloc[i_row, 2])
        response_options = response_options.append(i_row_response_options, sort=False,
    ignore_index=True)
    response_options.rename(columns={"id": "response_option_id"}, inplace=True)
```

```

    response_options[["survey_id", "page_id", "question_id", "response_option_id"]] =
response_options[
    ["survey_id", "page_id", "question_id", "response_option_id"]].astype(np.int64)

    # Store everything in a dictionary and return it.
    questionnaire_spec = {"pages": pages, "questions": questions, "items": items,
"response_options": response_options}
    return questionnaire_spec

```

2.2 Paradata import

```

# PARADATA IMPORT
# This code imports paradata from raw files exported from lKA.

# A majority of merging and linking between datafiles is not performed at this stage. The
exception include merging some
# information from the questionnaire metadata and pre-requirements for data cleaning and
handling of missing values
# at later stages.

# It is probably good idea to define function to import each paradata type. This is not done
in this script but should
# be quite easy to do. In this case, processing settings should be defined at the level of
individual functions.
# This file needs to be run second, after metadata parser. Code contained in other files
assume existence of some objects
# created by this code.

import pandas as pd
import numpy as np

# Define variables to store different types of paradata, settings and processing logs.
settings = {}
paradata_raw = {}
ref_data = {}
processing_log = []

# Source parameters.
settings["data_folder"] = "/Users/nejc/OneDrive - Univerza v Ljubljani/RR Projekti/2018
Paradata TP BL/Survey/Paradata/Raw/Paradata export SI rev0-NB/"
settings["metadata_file"] = "/Users/nejc/OneDrive - Univerza v Ljubljani/RR Projekti/2018
Paradata TP BL/Survey/Paradata/Ref/Master questionnaire SI export rev0-NB.json"
settings["item_versions_file"] = "/Users/nejc/OneDrive - Univerza v Ljubljani/RR Projekti/2018
Paradata TP BL/Survey/Paradata/Ref/Common item names SI rev0-GČ-NB.csv"
settings["survey_id"] = 248758

# UAS corrections file.
# Specify the location of a file to correct wrongly identified device info from uas. It needs
to contain the following
# columns:
# - search_string : uas string or part of it to search for in paradata;
# - device_brand_corr: manufacturer name
# - device_model_corr: device model name or code
# - is_mobile_corr: bool whether mobile phone
# - is_tablet_corr: bool whether tablet
# - is_pc_corr: bool whether pc
# - match_type: type of match for info correction (full_uas | full_model_code |
partial_model_code | manual)
settings["uas_corrections_file"] = "/Users/nejc/OneDrive - Univerza v Ljubljani/RR
Projekti/2018 Paradata TP BL/Survey/Paradata/Ref/UAS corrections rev0-NB.csv"

# TODO: Ideas for additional paradata processing settings:
# - Use response and messages data to impute missing pageviews identifiers (currently always
performed);
# - additional option to replace page IDs using questionnaire metadata in case of conflict
(currently always performed).
# - Which sources to use when imputing pageviews identifiers (currently uses UAS and
response/messages data).
# - Specify whether only one UAS per respondent should be enforced. This is useful when we
know that the respondent
# - could not terminate the participation and return later to finish it using another
device/browser.

```

```

# - Use UAS correction file (currently always used and assumed it exists)
# - Merge pageview entries into one if no other events were logged during the pageview
(currently yes)

# %% SURVEY METADATA
# Parse questionnaire metadata. Note that the script is only checked for question types and
formats used in this
# survey. Other surveys may contain unsupported types.
survey_metadata = metadata_parse(settings["metadata_file"])

# Because response_option_id is not always unique (e.g. for grid questions it equals value),
we replace it by
# putting together question_id and response_option_id.
survey_metadata["response_options"]["response_option_unique_id"] =
survey_metadata["response_options"][
    "question_id"].astype(str) + \

survey_metadata["response_options"][
    "response_option_id"].astype(str)

# Create sequential page identifier.
# TODO: Zaradi konsistentnosti to shrani kot ref_ objekt in ne v survey_metadata.
survey_metadata["pages"]["page_seq"] = survey_metadata["pages"].index
survey_metadata["pages"]["page_seq"] = np.where(survey_metadata["pages"]["type"] == "intro",
0, survey_metadata["pages"]["page_seq"])
survey_metadata["pages"]["page_seq"] = np.where(survey_metadata["pages"]["type"] == "end",
999999, survey_metadata["pages"]["page_seq"])

# %% PAGEVIEWS
# Import data and rename columns.
pageviews_raw = pd.read_csv(
    settings["data_folder"] + "advanced_paradata_pageviews_" + str(settings["survey_id"]) +
    ".csv",
    quotechar='"')
pageviews_raw.rename(columns={
    "id": "pageview_id",
    "ank_id": "survey_id",
    "gru_id": "page_id",
    "usr_id": "respondent_id",
    "load_time": "load_ts",
    "post_time": "post_ts",
    "user_agent": "uas",
    "language": "survey_lang",
    "devicePixelRatio": "pixel_ratio",
    "width": "screen_width",
    "height": "screen_height",
    "availWidth": "screen_width_avail",
    "availHeight": "screen_height_avail",
    "jquery_windowW": "viewport_width",
    "jquery_windowH": "viewport_height",
    "jquery_documentW": "page_width",
    "jquery_documentH": "page_height"},
    inplace=True, errors="raise")

# Convert timestamps into datetime format.
pageviews_raw["load_ts"] = pd.to_datetime(pageviews_raw["load_ts"],
    format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
pageviews_raw["post_ts"] = pd.to_datetime(pageviews_raw["post_ts"],
    format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")

# Check and log potential non-existing timestamps.
# IMPORTANT: Post timestamps were not used in this study and are therefore not addressed in
the code.
if pageviews_raw["load_ts"].isna().any():
    processing_log.append(
        ["pageviews_raw", "potential data loss", "Some pageview entries do not have valid
load_ts value.",
        pageviews_raw[pageviews_raw["load_ts"].isna()]])

# Check for known invalid page_id values and set them to missing.
if ((pageviews_raw["page_id"] == 0) | (pageviews_raw["respondent_id"] == 0)).any():
    processing_log.append(["pageviews_raw", "missing/invalid data",
        "Some pageview entries do not have a valid page ID. These IDs were
set to missing.",

```



```

        pageviews_raw[(pageviews_raw["page_id"] == 0) |
(pageviews_raw["respondent_id"] == 0)])

    pageviews_raw.replace({"page_id": {0: np.nan}, "respondent_id": {0: np.nan}, "recnum": {0:
np.nan}},
                           inplace=True)

# Check for other known invalid values and set them to missing.
# TODO: This would be good to log, too.
pageviews_raw.replace({"pixel_ratio": {0: np.nan},
                        "screen_width": {0: np.nan},
                        "screen_height": {0: np.nan},
                        "screen_width_avail": {0: np.nan},
                        "screen_height_avail": {0: np.nan},
                        "viewport_width": {0: np.nan},
                        "viewport_height": {0: np.nan},
                        "page_width": {0: np.nan},
                        "page_height": {0: np.nan}}, inplace=True)

# %% RESPONSES

# Import data and rename columns.
responses_raw = pd.read_csv(
    settings["data_folder"] + "advanced_paradata_responses_" + str(settings["survey_id"]) +
    ".csv",
    quotechar='"')
responses_raw.rename(columns={"id": "response_id",
                              "page_id": "pageview_id",
                              "spr_id": "spr_id_tmp",
                              "vre_id": "vre_id_tmp",
                              "time": "response_ts",
                              "event": "event_tmp",
                              "value": "value_tmp"},
                    inplace=True, errors="raise")

# Check for entries with known invalid identifier(s). They need to be removed to prevent
errors when merging
# with metadata.
if (responses_raw["spr_id_tmp"] == 0).any():
    processing_log.append(["responses_raw", "data deletion",
                          "Some entries do not have valid response ID and were removed to
prevent errors in further "
                          "processing.",
                          responses_raw[responses_raw["spr_id_tmp"] == 0]])
    responses_raw = responses_raw[responses_raw["spr_id_tmp"] != 0]

# Create a unique identifier for response options to match with metadata.
responses_raw["spr_vre_id_tmp"] = responses_raw["spr_id_tmp"].astype(str) +
responses_raw["vre_id_tmp"].astype(str)

# %% Derivation of question info from the survey metadata
# Process each question type in a separate dataframe. Basic identifiers and key info (e.g.
name and value) are
# obtained at this stage as well.
# IMPORTANT: This is not a general solution and may not work for question types that were not
used in this
# questionnaire.

# Get question names and types by merging the questions metadata..
responses_raw = pd.merge(responses_raw, survey_metadata["questions"][["question_id", "name",
"type", "input"]],
                        left_on="spr_id_tmp", right_on="question_id", how="left")
responses_raw.rename(columns={"name": "question_name",
                              "type": "question_type",
                              "input": "question_input_type"}, inplace=True, errors="raise")

# Single answer radio:
# - get question_id from spr_id_tmp
# - because this is a single-item question, item_id equals question_id
# - get response_option_id from vre_id_tmp
responses_raw_single_answer = responses_raw[responses_raw["question_type"] ==
"single_answer"].copy()
responses_raw_single_answer["item_id"] = responses_raw_single_answer["question_id"]
responses_raw_single_answer["item_name"] = responses_raw_single_answer["question_name"]
responses_raw_single_answer = pd.merge(responses_raw_single_answer,

```

```

survey_metadata["response_options"][["response_option_id",
"response_option_unique_id",
                                "value"]],
                                left_on="vre_id_tmp", right_on="response_option_id",
how="left",
                                indicator="response_option_merge_status")
responses_raw_single_answer.rename(columns={"value": "response_option_code"}, inplace=True)

# Multiple answer checkbox
# - get question_id from spr_id_tmp
# - get item_id from vre_id_tmp for normal items
# - get response_option_id from vre_id_tmp for nonsubstantive response options
# - in case of nonsubstantive response options item_id and item_name equals question id and
name.
responses_raw_multi_answer = responses_raw[responses_raw["question_type"] ==
"multi_answer"].copy()
responses_raw_multi_answer = pd.merge(responses_raw_multi_answer,
survey_metadata["items"][["item_id", "name"]],
                                left_on="vre_id_tmp", right_on="item_id", how="left",
                                indicator="item_merge_status")
responses_raw_multi_answer = pd.merge(responses_raw_multi_answer,
survey_metadata["response_options"][["response_option_id",
"response_option_unique_id",
                                "value"]],
                                left_on="vre_id_tmp", right_on="response_option_id",
how="left",
                                indicator="response_option_merge_status")
responses_raw_multi_answer.rename(columns={"name": "item_name",
"value": "response_option_code"}, inplace=True,
errors="raise")
responses_raw_multi_answer["item_id"] = np.where(responses_raw_multi_answer["item_id"].isna(),
responses_raw_multi_answer["question_id"],
responses_raw_multi_answer["item_id"])
responses_raw_multi_answer["item_name"] =
np.where(responses_raw_multi_answer["item_name"].isna(),
responses_raw_multi_answer["question_name"],
responses_raw_multi_answer["item_name"])

# Open numeric
# - get question_id from spr_id_tmp
# - get answer_id from spr_vre_id_tmp (only for nonsubstantive if any; needs to use created
unique identifier
# due to repeated answer codes).
# - get item_id from items metadata by matching question_id. # IMPORTANT: This only works
because we have no
# multiple-item open-ended responses in this survey (i.e. we always have only one input
field in open
# questions).
responses_raw_open = responses_raw[responses_raw["question_type"] == "open"].copy()
responses_raw_open = pd.merge(responses_raw_open, survey_metadata["items"][["question_id",
"item_id", "name"]],
                                left_on="spr_id_tmp", right_on="question_id", how="left",
                                indicator="item_merge_status")
responses_raw_open = pd.merge(responses_raw_open,
survey_metadata["response_options"][["response_option_id",
"response_option_unique_id",
"value"]],
                                left_on="spr_vre_id_tmp", right_on="response_option_unique_id",
how="left",
                                indicator="response_option_merge_status")
responses_raw_open.rename(columns={"name": "item_name",
"value": "response_option_code"}, inplace=True,
errors="raise")
# Check if question_id from initial and merged database match. If yes, use only one. Otherwise
log an
# error and stop execution.
if (responses_raw_open["question_id_x"] != responses_raw_open["question_id_y"]).sum() == 0:
    responses_raw_open.rename(columns={"question_id_x": "question_id"}, inplace=True)
    responses_raw_open.drop("question_id_y", axis=1, inplace=True)

```

```

else:
    processing_log.append(["responses_raw", "data inconsistency",
                          "Conflicting questions IDs while merging item metadata for open
numeric questions.",
                          responses_raw_open[
                              (responses_raw_open["question_id_x"] !=
responses_raw_open["question_id_y"])]])
    raise ValueError('Conflicting questions IDs while merging item metadata for open numeric
questions.')
```

Grid

- # - get item_id from spr_id_tmp
- # - get question_id from item_id and match missing question info
- # - get answer_id from spr_vre_id_tmp (needs to use the created unique identifier)

First we drop columns that are empty and will be filled by merging info from other data.
This removes the need
to rename equally named columns from merged dataframes later.

```

responses_raw_grids = responses_raw[responses_raw["question_type"].isna()].copy()
responses_raw_grids.drop(["question_id", "question_name", "question_type",
"question_input_type"], axis=1,
                          inplace=True)
responses_raw_grids["item_id"] = responses_raw_grids["spr_id_tmp"]
responses_raw_grids = pd.merge(responses_raw_grids, survey_metadata["items"][["question_id",
"item_id", "name"]],
                              on="item_id", how="left", indicator="item_merge_status")
responses_raw_grids.rename(columns={"name": "item_name"}, inplace=True, errors="raise")
responses_raw_grids = pd.merge(responses_raw_grids,
                              survey_metadata["questions"][["question_id", "name", "type",
"input"]],
                              on="question_id", how="left",
indicator="question_merge_status")
responses_raw_grids.rename(columns={"name": "question_name",
                                  "type": "question_type",
                                  "input": "question_input_type"}, inplace=True,
errors="raise")
```

Response_option_unique_id needs to be redefined, because spr_vre_id_tmp does not contain
question id with grid

questions.

```

responses_raw_grids["response_option_unique_id"] =
responses_raw_grids["question_id"].astype(str) + \

responses_raw_grids["vre_id_tmp"].astype(str)
responses_raw_grids = pd.merge(responses_raw_grids,
survey_metadata["response_options"][["response_option_id",

"response_option_unique_id",

"value"]],
                              on="response_option_unique_id", how="left",
indicator="response_option_merge_status")
responses_raw_grids.rename(columns={"value": "response_option_code"}, inplace=True)
```

Replace the original dataframe with edited data and tidy up.

```

responses_raw = responses_raw_single_answer.append(
    [responses_raw_multi_answer, responses_raw_open, responses_raw_grids])
responses_raw.drop(["item_merge_status", "question_merge_status",
"response_option_merge_status"], axis=1,
                    inplace=True)
```

Create a new variable describing response action performed (instead of having two
variables).

```

responses_raw["response_action"] = "undefined"
responses_raw["response_action"] = np.where(
    (responses_raw["event_tmp"] == "radio_checkbox_change") & (responses_raw["value_tmp"] ==
"1"),
    "selected", responses_raw["response_action"])
responses_raw["response_action"] = np.where(
    (responses_raw["event_tmp"] == "radio_checkbox_change") & (responses_raw["value_tmp"] ==
"0"),
    "unselected", responses_raw["response_action"])
responses_raw["response_action"] = np.where(
    (responses_raw["event_tmp"] == "text_enter") | (responses_raw["event_tmp"] ==
"text_leave"),
    responses_raw["event_tmp"], responses_raw["response_action"])
responses_raw["response_value"] = np.where(
```

```

        (responses_raw["event_tmp"] == "text_enter") | (responses_raw["event_tmp"] ==
"text_leave"),
        responses_raw["value_tmp"], responses_raw["response_option_code"])
responses_raw["response_value"] = np.where(
    (responses_raw["response_action"] == "selected") & (responses_raw["question_type"] ==
"multi_answer") & (
        responses_raw["question_input_type"] == "checkbox"),
    "1", responses_raw["response_value"])
responses_raw["response_value"] = np.where(
    (responses_raw["response_action"] == "unselected") & (responses_raw["question_type"] ==
"multi_answer") & (
        responses_raw["question_input_type"] == "checkbox"),
    "0", responses_raw["response_value"])

# %% Finalisation
# Convert timestamps into datetime format and check for missing timestamps.
responses_raw["response_ts"] = pd.to_datetime(responses_raw["response_ts"],
                                              format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if responses_raw["response_ts"].isna().any():
    processing_log.append(["responses_raw", "potential data loss",
        "Some response entries do not have valid response_ts value.",
        responses_raw[responses_raw["response_ts"].isna()]])

# Add page ids. This is done at this stage using metadata to enable potential imputation of
missing page ids
# during the processing of pageviews data.
responses_raw = pd.merge(responses_raw, survey_metadata["questions"][["question_id",
"page_id"]],
                        on="question_id", how="left", indicator="page_id_merge")
if (responses_raw["page_id_merge"] != "both").any():
    processing_log.append(["responses_raw", "unmerged entries",
        "Some response entries were not assigned page_id value from the
questionnaire metadata.",
        responses_raw[responses_raw["page_id_merge"] != "both"]])
responses_raw.rename(columns={"page_id": "page_id_meta"}, inplace=True, errors="raise")

# Keep only relevant variables, reordered column and change identifiers to int if none is
missing.
responses_raw = responses_raw[["response_id", "pageview_id", "response_ts", "question_id",
"question_name",
                                "question_type", "question_input_type", "item_id", "item_name",
                                "response_option_id", "response_option_unique_id",
"response_option_code",
                                "response_action", "response_value", "page_id_meta"]]
responses_raw["question_id"] = responses_raw["question_id"].astype(np.int64)

# Drop unneeded objects.
del responses_raw_grids
del responses_raw_multi_answer
del responses_raw_open
del responses_raw_single_answer

# %% MESSAGES

# Import data and rename columns.
messages_raw = pd.read_csv(settings["data_folder"] + "advanced_paradata_msg_" +
str(settings["survey_id"]) + ".csv",
                           quotechar='"')
messages_raw.rename(columns={"id": "msg_id",
                            "page_id": "pageview_id",
                            "time_display": "display_ts",
                            "time_close": "close_ts",
                            "type": "type",
                            "trigger_id": "trigger_question_id",
                            "trigger_type": "trigger_type",
                            "ignorable": "ignorable",
                            "text": "msg_text",
                            "action": "respondent_action"},
                    inplace=True, errors="raise")

# Merge with question info.
messages_raw = pd.merge(messages_raw, survey_metadata["questions"][["question_id", "name",
"type"]],
                        left_on="trigger_question_id", right_on="question_id", how="left",
                        indicator="questions_merge")
if (messages_raw["questions_merge"] != "both").any():

```

```

        processing_log.append(["messages_raw", "unmerged entries",
                               "Some response message entries were not assigned question info from
the questionnaire "
                               "metadata.",
                               messages_raw[messages_raw["questions_merge"] != "both"]])
messages_raw.drop(columns="question_id", inplace=True)

# Identify message type and check that no exceptions occurred. Multiple original variables are
used to assure
# that all data are consistently logged. Note that several conditions need to be used, because
the "ignorable"
# variable provided by lKA is not valid for all message types (e.g. numeric validations).
messages_raw["ignorable"] = np.where(messages_raw["ignorable"] == 1, True, False)
messages_raw["msg_type"] = "undefined"
messages_raw["msg_type"] = np.where(
    (messages_raw["trigger_type"] == "sof") | (messages_raw["trigger_type"] == "har"),
    "item_nonresponse",
    messages_raw["msg_type"])
messages_raw["msg_type"] = np.where(messages_raw["trigger_type"] == "num", "invalid_number",
                                   messages_raw["msg_type"])
if (messages_raw["msg_type"] == "undefined").any():
    processing_log.append(
        ["messages_raw", "missing/invalid data", "Some response messages are of unknown
type.",
        messages_raw[messages_raw["msg_type"] == "undefined"]])

# Convert timestamps to datetime format and check for missing timestamps.
messages_raw["display_ts"] = pd.to_datetime(messages_raw["display_ts"],
                                             format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if messages_raw["display_ts"].isna().any():
    processing_log.append(["messages_raw", "potential data loss",
                           "Some response message entries do not have valid display_ts
value.",
                           messages_raw[messages_raw["display_ts"].isna()]])

messages_raw["close_ts"] = pd.to_datetime(messages_raw["close_ts"],
                                           format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if messages_raw["close_ts"].isna().any():
    processing_log.append(["messages_raw", "potential data loss",
                           "Some response message entries do not have valid close_ts value.",
                           messages_raw[messages_raw["close_ts"].isna()]])

# Add page ids. This is done at this stage using metadata to enable potential imputation of
missing page ids
# during the processing of pageviews data.
messages_raw = pd.merge(messages_raw, survey_metadata["questions"][["question_id",
"page_id"]],
                        left_on="trigger_question_id", right_on="question_id", how="left",
                        indicator="page_id_merge")
if (messages_raw["page_id_merge"] != "both").any():
    processing_log.append(["messages_raw", "unmerged entries",
                           "Some response message entries were not assigned page_id value from
the questionnaire metadata.",
                           messages_raw[messages_raw["page_id_merge"] != "both"]])

# Rename and reorder columns and keep only relevant variables.
messages_raw.rename(columns={"name": "trigger_question_name",
                             "page_id": "page_id_meta"}, inplace=True, errors="raise")
messages_raw = messages_raw[["msg_id", "pageview_id", "msg_type", "display_ts", "close_ts",
"trigger_question_id",
                             "trigger_question_name", "ignorable", "msg_text",
"respondent_action",
                             "page_id_meta"]]

# %% POINTER MOVEMENT

# Import data and rename columns.
pointer_raw = pd.read_csv(settings["data_folder"] + "advanced_paradata_pointer_" +
str(settings["survey_id"]) + ".csv",
                           quotechar="'")
pointer_raw.rename(columns={"id": "move_id",
                             "page_id": "pageview_id",
                             "time_start": "move_start_ts",
                             "time_end": "move_end_ts",
                             "pos_x_start": "start_coord_x",
                             "pos_y_start": "start_coord_y",

```

```

        "pos_x_end": "end_coord_x",
        "pos_y_end": "end_coord_y",
        "distance": "move_distance"},
        inplace=True, errors="raise")

# Convert timestamps to datetime format and check for missing timestamps.
pointer_raw["move_start_ts"] = pd.to_datetime(pointer_raw["move_start_ts"],
        format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
pointer_raw["move_end_ts"] = pd.to_datetime(pointer_raw["move_end_ts"],
        format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if pointer_raw["move_start_ts"].isna().any():
    processing_log.append(["pointer_raw", "potential data loss",
        "Some pointer movement events do not have valid move_start_ts
value.",
        pointer_raw[pointer_raw["move_start_ts"].isna()]])
if pointer_raw["move_end_ts"].isna().any():
    processing_log.append(["pointer_raw", "potential data loss",
        "Some pointer movement events do not have valid move_end_ts
value.",
        pointer_raw[pointer_raw["move_end_ts"].isna()]])

# %% OTHER PARADATA
# This bit only imports the whole "other" dataset as a temporary object that is later used to
create several datasets
# containing different types of paradata.

# Import data and rename columns.
other_raw = pd.read_csv(
    settings["data_folder"] + "advanced_paradata_other_" + str(settings["survey_id"]) +
    ".csv",
    quotechar='"')

# %% PAGE SCROLLING

# Take relevant events from the dataset and rename columns.
page_scrolls_raw = other_raw.loc[other_raw["event"] == "scroll_page", ["id", "page_id",
"time", "pos_x", "pos_y"]].copy()
page_scrolls_raw.rename(columns={"id": "scroll_id",
        "page_id": "pageview_id",
        "time": "scroll_ts",
        "pos_x": "scroll_coord_x",
        "pos_y": "scroll_coord_y"},
        inplace=True, errors="raise")

# Convert timestamps to datetime format and check for missing timestamps.
page_scrolls_raw["scroll_ts"] = pd.to_datetime(page_scrolls_raw["scroll_ts"],
        format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if page_scrolls_raw["scroll_ts"].isna().any():
    processing_log.append(["page_scrolls_raw", "potential data loss",
        "Some scroll events do not have valid scroll_ts value.",
        page_scrolls_raw[page_scrolls_raw["scroll_ts"].isna()]])

# %% CLICK

# Take relevant events from the dataset and rename columns.
clicks_raw = other_raw.loc[
    other_raw["event"] == "click", ["id", "page_id", "time", "pos_x", "pos_y", "div_id",
"div_class", "div_type"]].copy()
clicks_raw.rename(columns={"id": "click_id",
        "page_id": "pageview_id",
        "time": "click_ts",
        "pos_x": "coord_x",
        "pos_y": "coord_y"},
        inplace=True, errors="raise")

# Convert timestamps to datetime format and check for missing timestamps.
clicks_raw["click_ts"] = pd.to_datetime(clicks_raw["click_ts"],
        format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if clicks_raw["click_ts"].isna().any():
    processing_log.append(["clicks_raw", "potential data loss",
        "Some scroll events do not have valid click_ts value.",
        clicks_raw[clicks_raw["click_ts"].isna()]])

# %% FOCUS CHANGES
# Take relevant events from the dataset and rename columns.
focus_changes_raw = other_raw.loc[

```

```

        (other_raw["event"] == "focus") | (other_raw["event"] == "blur"),
        ["id", "page_id", "time", "event"]).copy()
focus_changes_raw.rename(columns={"id": "focus_change_id",
                                   "page_id": "pageview_id",
                                   "time": "focus_change_ts",
                                   "event": "change_type"},
                           inplace=True, errors="raise")

# Convert timestamps to datetime format and check for missing timestamps.
focus_changes_raw["focus_change_ts"] = pd.to_datetime(focus_changes_raw["focus_change_ts"],
                                                         format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if focus_changes_raw["focus_change_ts"].isna().any():
    processing_log.append(["focus_changes_raw", "potential data loss",
                          "Some scroll events do not have valid focus_change_ts value.",
                          focus_changes_raw[focus_changes_raw["focus_change_ts"].isna()]])

# %% VIEW CHANGES
# Take relevant events from the dataset and rename columns.
view_changes_raw = other_raw.loc[
    (other_raw["event"] == "resize") | (other_raw["event"] == "pinch_resize") |
    (other_raw["event"] == "orientation_change"),
    ["id", "page_id", "time", "event", "value", "pos_x", "pos_y"]].copy()
# TODO Check the names of resolution variables and rename them if needed.
view_changes_raw.rename(columns={"id": "view_change_id",
                                   "page_id": "pageview_id",
                                   "time": "view_change_ts",
                                   "event": "change_type",
                                   "value": "value_field_1",
                                   "pos_x": "res_h",
                                   "pos_y": "res_v"},
                           inplace=True, errors="raise")

# In raw data we only change the name of events to make it clearer for further processing.
view_changes_raw["change_type"] = np.where(view_changes_raw["change_type"] == "resize",
                                             "zoom_or_window_resize",
                                             view_changes_raw["change_type"])
view_changes_raw["change_type"] = np.where(view_changes_raw["change_type"] == "pinch_resize",
                                             "pinch_zoom",
                                             view_changes_raw["change_type"])

# Convert timestamps to datetime format and check for missing timestamps.
view_changes_raw["view_change_ts"] = pd.to_datetime(view_changes_raw["view_change_ts"],
                                                         format="%Y-%m-%d %H:%M:%S.%f", errors="coerce")
if view_changes_raw["view_change_ts"].isna().any():
    processing_log.append(["view_changes_raw", "potential data loss",
                          "Some scroll events do not have valid view_change_ts value.",
                          view_changes_raw[view_changes_raw["view_change_ts"].isna()]])

# %% ALL EVENTS LOG
# The all events log contains timestamps and types of logged events. This is used for
# diagnostics and identification
# of inactivity times.
# Events that have multiple timestamps logged as part of the same record (e.g. start and end
# time in one row) are
# split into several events. Note that IDs of such events are repeated (i.e. not all event_id
# values are unique).
# Note page post should be added, but was not recorded in this survey.
# TODO: Include events from other.

event_list_raw = pd.concat([pageviews_raw[["pageview_id", "load_ts"]],
                            responses_raw[["response_id", "pageview_id", "response_ts"]],
                            messages_raw[["msg_id", "pageview_id", "display_ts"]],
                            messages_raw[["msg_id", "pageview_id", "close_ts"]],
                            pointer_raw[["move_id", "pageview_id", "move_start_ts"]],
                            pointer_raw[["move_id", "pageview_id", "move_end_ts"]],
                            page_scrolls_raw[["scroll_id", "pageview_id", "scroll_ts"]],
                            clicks_raw[["click_id", "pageview_id", "click_ts"]],
                            focus_changes_raw[["focus_change_id", "pageview_id",
                                                "focus_change_ts", "change_type"]],
                            view_changes_raw[["view_change_id", "pageview_id",
                                                "view_change_ts", "change_type"]]])

# Create unified event ID, event type and event timestamp variables. The code below takes
# advantage of having uniquely

```

```

# named timestamp variables in each dataframe.
event_list_raw["event_id"] = np.where(event_list_raw["load_ts"].notna(),
event_list_raw["pageview_id"], np.nan)
event_list_raw["event_type"] = np.where(event_list_raw["load_ts"].notna(), "pageview", np.nan)
event_list_raw["event_ts"] = np.where(event_list_raw["load_ts"].notna(),
event_list_raw["load_ts"], pd.NaT)

event_list_raw["event_id"] = np.where(event_list_raw["response_ts"].notna(),
event_list_raw["response_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["response_ts"].notna(), "response",
event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["response_ts"].notna(),
event_list_raw["response_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["display_ts"].notna(),
event_list_raw["msg_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["display_ts"].notna(),
"message_display", event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["display_ts"].notna(),
event_list_raw["display_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["close_ts"].notna(),
event_list_raw["msg_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["close_ts"].notna(), "message_close",
event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["close_ts"].notna(),
event_list_raw["close_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["move_start_ts"].notna(),
event_list_raw["move_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["move_start_ts"].notna(),
"pointer_move_start", event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["move_start_ts"].notna(),
event_list_raw["move_start_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["move_end_ts"].notna(),
event_list_raw["move_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["move_end_ts"].notna(),
"pointer_move_end", event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["move_end_ts"].notna(),
event_list_raw["move_end_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["scroll_ts"].notna(),
event_list_raw["scroll_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["scroll_ts"].notna(), "scroll",
event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["scroll_ts"].notna(),
event_list_raw["scroll_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["click_ts"].notna(),
event_list_raw["click_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["click_ts"].notna(), "click",
event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["click_ts"].notna(),
event_list_raw["click_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["focus_change_ts"].notna(),
event_list_raw["focus_change_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where((event_list_raw["focus_change_ts"].notna()) &
(event_list_raw["change_type"] == "focus"),
"window_focus", event_list_raw["event_type"])
event_list_raw["event_type"] = np.where((event_list_raw["focus_change_ts"].notna()) &
(event_list_raw["change_type"] == "blur"),
"window_blur", event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["focus_change_ts"].notna(),
event_list_raw["focus_change_ts"], event_list_raw["event_ts"])

event_list_raw["event_id"] = np.where(event_list_raw["view_change_ts"].notna(),
event_list_raw["view_change_id"], event_list_raw["event_id"])
event_list_raw["event_type"] = np.where(event_list_raw["view_change_ts"].notna(),
event_list_raw["change_type"], event_list_raw["event_type"])
event_list_raw["event_ts"] = np.where(event_list_raw["view_change_ts"].notna(),
event_list_raw["view_change_ts"], event_list_raw["event_ts"])

event_list_raw["event_ts"] = event_list_raw["event_ts"].astype(np.datetime64)

```


2.3 Paradata processing

```
# PARADATA PROCESSING
# This code takes imported paradata and performs various data processing operations to clean,
# remove duplicates and
# invalid data, handle missings etc. Its outputs are used as input for paradata analysis.

# It is probably good idea to define function to process each paradata type. This is not done
# in this script but should
# be quite easy to do. In this case, processing settings should be defined at the level of
# individual functions.
# This file needs to be run second, after metadata parser. Code contained in other files
# assume existance of some objects
# created by this code.

import pandas as pd
import numpy as np
import user_agents as ua

# This check is therefore
# only perform for non-missing data in page_id and page_id_meta. Depending on the settings,
# conflicting
# page_id_meta values are either set to missing or used to replace the existing page_id values

# %% PAGEVIEWS
pageviews_proc = pageviews_raw.copy()

# %% Imputation of missing identifier data
# TODO: Check that the required tables of raw response and messages data exist. Messages data
# are useful for
# cases where no response was provided, triggering item nonresponse reminder. This is only
# applicable for pages
# containing at least one question with nonresponse reminder (or any other kind of validation)
# and pageviews
# during which such validation was triggered.

# STAGE 1: Impute page ids based on responses and messages data.
# Create a list for matching pageviews and page ids from raw responses and messages data.
pageview_page_ids = responses_raw[["pageview_id", "page_id_meta"]].drop_duplicates()
pageview_page_ids = pageview_page_ids.append(messages_raw[["pageview_id",
"page_id_meta"]]).drop_duplicates()
pageview_page_ids.drop_duplicates(inplace=True)

# Check that all pageview ids are unique
if pageview_page_ids["pageview_id"].is_unique == False:
    processing_log.append(["pageviews_proc", "inconsistent value",
                          "Not all pageviews IDs used for imputation of missing identifiers
are unique. "
                          "This may cause errors in the script performance and indicate
underlying problems.",
                          ])

pageview_page_ids[pageview_page_ids["pageview_id"].duplicated(keep=False) == True]]

# Merge page_id_meta with pageview entries and check for consistency against the existing page
# ids. This is done
# to verify the consistency of data used for imputation. Note that missings may occur in
# page_id_meta in various
# scenarios, e.g. when the respondent goes back and does not change any response as this will
# trigger no
# response entries or INR reminder entries which are used to derive page_id_meta values.
pageviews_proc = pd.merge(pageviews_proc, pageview_page_ids, how="left", on="pageview_id",
                          indicator="page_id merge")
if ((pageviews_proc["page_id"].notnull() & (pageviews_proc["page_id_meta"].notnull() &
(pageviews_proc["page_id"] != pageviews_proc["page_id_meta"])).any():
    processing_log.append(
        ["pageviews_proc", "inconsistent identifiers",
        "For some pageviews entries there is a mismatch between existing page_ids and
page_id_meta values used for "
        "imputation. This may be caused by changes in survey page breaks during the data "
        "collection. The original page_ids for these cases will be replaced by page_id_meta "
        "values, existing values are logged.",
        pageviews_proc[pageviews_proc["page_id"].notnull() &
        (pageviews_proc["page_id_meta"].notnull() &
```

```

        (pageviews_proc["page_id"] != pageviews_proc["page_id_meta"])]])
pageviews_proc["page_id"] = np.where(((pageviews_proc["page_id"].notnull()) &
                                     (pageviews_proc["page_id_meta"].notnull()) &
                                     (pageviews_proc["page_id"] !=
pageviews_proc["page_id_meta"]))),
                                     pageviews_proc["page_id_meta"],
pageviews_proc["page_id"])

pageviews_proc["page_id"].fillna(pageviews_proc["page_id_meta"], inplace=True)

del pageview_page_ids

# "Manually" correct page IDs of pages containing only captions that were not identified
above. This is due to changes in page breaks
# that were done during the data collection. However, some changes may still be missed :-/
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511226),
                                     12511228, pageviews_proc["page_id"])
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511205),
                                     12511206, pageviews_proc["page_id"])
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511248),
                                     12511253, pageviews_proc["page_id"])
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511256),
                                     12511261, pageviews_proc["page_id"])

processing_log.append(["pageviews_proc", "manual correction of identifiers",
                    """Some page identifiers were manually corrected. The complete source
code is provided here:
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511226),
                                     12511228, pageviews_proc["page_id"])
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511205),
                                     12511206, pageviews_proc["page_id"])
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511248),
                                     12511253, pageviews_proc["page_id"])
pageviews_proc["page_id"] = np.where((pageviews_proc["recnum"] >= 50) &
                                     (pageviews_proc["recnum"] <= 101) & (pageviews_proc["page_id"] == 12511256),
                                     12511261, pageviews_proc["page_id"])"""])

# STAGE 2: Impute respondent IDs and the remaining missing page IDs.
# Create a list of entries with missing ID.
imputed_ids = pageviews_proc.loc[
pageviews_proc["respondent_id"].isnull(), ["pageview_id", "respondent_id", "load_ts", "uas"]]
imputed_ids.sort_values("load_ts", inplace=True)

# Identify potential imputation sources as entries with nonmissing identifiers that have the
matching UAS and
# were present in the survey during the hour when any missings with such UA occurred. First
the matching
# candidate respondents are identified, then pageviews entries of these respondents are
selected to find the nearest
# pageview entry candidate to be used for imputation.
resp_id_sources = pageviews_proc.loc[
pageviews_proc["respondent_id"].notna(), ["respondent_id", "load_ts", "uas"]].copy()
resp_id_sources["hr_uas"] = resp_id_sources["load_ts"].dt.strftime("%Y%m%d%H") + "_" + \
    resp_id_sources["uas"]
resp_id_sources = resp_id_sources.groupby(["respondent_id", "uas"])["load_ts"].agg(
["min", "max"]).reset_index()

def impute_pageview_ids(i_row):
    i_date = i_row["load_ts"]
    i_uas = i_row["uas"]
    # Check the list of potential imputation sources for match on both: UAS and the period
between the first and
    # the last page load event covering the time when the page load event with missing ID
occurred.
    i_candidate_respondents = resp_id_sources[
        (resp_id_sources["uas"] == i_uas) & (resp_id_sources["min"] < i_date) & (
            resp_id_sources["max"] > i_date)]
    i_n_unique = i_candidate_respondents["respondent_id"].nunique()

```

```

        if i_n_unique == 1:
            # If only one respondent ID matches the UAS and the period during which the event with
            # missing ID occurred,
            # the nearest previous pageview by this respondent is used to provide the data for the
            # missing entry.
            i_candidate_pageviews = pageviews_proc[

pageviews_proc["respondent_id"].isin(i_candidate_respondents["respondent_id"])).copy()
            i_candidate_pageviews = i_candidate_pageviews[
                (i_date - i_candidate_pageviews["load_ts"]).dt.total_seconds() > 0]
            i_delta_min = (i_date - i_candidate_pageviews["load_ts"]).min()
            i_row["imp_respondent_id"] = i_candidate_pageviews.loc[
                (i_date - i_candidate_pageviews["load_ts"]) == i_delta_min,
"respondent_id"].iloc[0]
            i_row["match"] = "single_match"
            i_row["no_matches"] = i_n_unique
            i_row["match_load_time_diff"] = i_delta_min.total_seconds()
            i_row["source_pageview_id"] = i_candidate_pageviews.loc[
                (i_date - i_candidate_pageviews["load_ts"]) == i_delta_min, "pageview_id"].iloc[0]
            i_row["imp_recnum"] = \
                pageviews_proc.loc[pageviews_proc["pageview_id"] == i_row["source_pageview_id"],
"recnum"].iloc[0]
            i_row["imp_page_id"] = \
                pageviews_proc.loc[pageviews_proc["pageview_id"] == i_row["source_pageview_id"],
"page_id"].iloc[0]
            elif i_n_unique > 1:
                # If pageload events with several different respondent ID entries match the UAS and
                # the period during which
                # the event with missing ID occurred, the entry with the nearest previous timestamp
                # among these is used to
                # provide the data for the missing entry.
                i_candidate_pageviews = pageviews_proc[

pageviews_proc["respondent_id"].isin(i_candidate_respondents["respondent_id"])).copy()
                i_candidate_pageviews = i_candidate_pageviews[
                    (i_date - i_candidate_pageviews["load_ts"]).dt.total_seconds() > 0]
                i_delta_min = (i_date - i_candidate_pageviews["load_ts"]).min()
                i_row["imp_respondent_id"] = i_candidate_pageviews.loc[
                    (i_date - i_candidate_pageviews["load_ts"]) == i_delta_min,
"respondent_id"].iloc[0]
                i_row["match"] = "nearest_TS_UAS"
                i_row["no_matches"] = i_n_unique
                i_row["match_load_time_diff"] = i_delta_min.total_seconds()
                i_row["source_pageview_id"] = i_candidate_pageviews.loc[
                    (i_date - i_candidate_pageviews["load_ts"]) == i_delta_min, "pageview_id"].iloc[0]
                i_row["imp_recnum"] = \
                    pageviews_proc.loc[pageviews_proc["pageview_id"] == i_row["source_pageview_id"],
"recnum"].iloc[0]
                i_row["imp_page_id"] = \
                    pageviews_proc.loc[pageviews_proc["pageview_id"] == i_row["source_pageview_id"],
"page_id"].iloc[0]
            else:
                # If no matches by UAS and the first-last timestamp period are found, the respondent
                # ID from the entry with
                # a matching UAS and the nearest previous timestamp is used.
                i_candidate_pageviews = pageviews_proc[
                    (pageviews_proc["respondent_id"].notna()) & (pageviews_proc["uas"] ==
i_uas)].copy()
                i_candidate_pageviews = i_candidate_pageviews[
                    (i_date - i_candidate_pageviews["load_ts"]).dt.total_seconds() > 0]
                i_delta_min = (i_date - i_candidate_pageviews["load_ts"]).min()
                if i_candidate_pageviews.empty == False:
                    i_delta_min = abs((i_candidate_pageviews["load_ts"] - i_date)).min()
                    i_row["imp_respondent_id"] = i_candidate_pageviews.loc[
                        (i_date - i_candidate_pageviews["load_ts"]) == i_delta_min,
"respondent_id"].iloc[0]
                    i_row["match"] = "nearest_UAS"
                    i_row["no_matches"] = i_n_unique
                    i_row["match_load_time_diff"] = i_delta_min.total_seconds()
                    i_row["source_pageview_id"] = i_candidate_pageviews.loc[
                        (i_date - i_candidate_pageviews["load_ts"]) == i_delta_min,
"pageview_id"].iloc[0]
                    i_row["imp_recnum"] = \
                        pageviews_proc.loc[pageviews_proc["pageview_id"] ==
i_row["source_pageview_id"], "recnum"].iloc[0]
                    i_row["imp_page_id"] = \

```

```

        pageviews_proc.loc[pageviews_proc["pageview_id"] ==
i_row["source_pageview_id"], "page_id"].iloc[0]
    else:
        i_row["imp_respondent_id"] = np.nan
        i_row["match"] = "no match"
        i_row["no_matches"] = np.nan
        i_row["match_load_time_diff"] = np.nan
        i_row["source_pageview_id"] = np.nan
        i_row["imp_recnum"] = np.nan
        i_row["imp_page_id"] = np.nan
    return i_row

# Impute missing IDs.
imputed_ids = imputed_ids.apply(impute_pageview_ids, axis=1)
pageviews_proc = pd.merge(pageviews_proc,
                           imputed_ids[["pageview_id", "imp_respondent_id", "imp_recnum",
"imp_page_id",
                           "match", "match_load_time_diff",
"source_pageview_id"]],
                           on="pageview_id", how="left")
pageviews_proc["respondent_id"] = np.where(pageviews_proc["respondent_id"].isna(),
                                           pageviews_proc["imp_respondent_id"],
                                           pageviews_proc["respondent_id"])
pageviews_proc["recnum"] = np.where(pageviews_proc["recnum"].isna(),
                                     pageviews_proc["imp_recnum"],
                                     pageviews_proc["recnum"])
pageviews_proc["page_id"] = np.where(pageviews_proc["page_id"].isna(),
                                     pageviews_proc["imp_page_id"],
                                     pageviews_proc["page_id"])

# Save the imputation info in the processing log.
processing_log.append(["pageviews_proc", "missing values imputation",
                      "Imputation of missing respondent and page identifiers in pageviews data
has been performed. "
                      "This log entry contains the list of potential replacements; actual
replacements have been "
                      "performed only where data was missing.", imputed_ids])

del imputed_ids
del resp_id_sources

# Handle the remaining entries with missing page/respondent IDs.
# Check whether any other paradata events occurred during these pageviews; for those we tried
to manually find the IDs. If ID could not
# be matched manually they were dropped along other remaining entries with missing IDs.
missing_ids = pageviews_proc[(pageviews_proc["respondent_id"].isna()) |
pageviews_proc["page_id"].isna()].copy()
missing_ids["has_events"] =
missing_ids["pageview_id"].isin(event_list_raw.loc[event_list_raw["event_type"] != "pageview",
"pageview_id"])

pageviews_proc["respondent_id"] = np.where(pageviews_proc["pageview_id"] == 12376, 30754752,
pageviews_proc["respondent_id"])
pageviews_proc["recnum"] = np.where(pageviews_proc["pageview_id"] == 12376, 154,
pageviews_proc["recnum"])
missing_ids = missing_ids[missing_ids["pageview_id"] != 12376]

pageviews_proc = pageviews_proc[(pageviews_proc["respondent_id"].notna()) &
pageviews_proc["page_id"].notna()]

processing_log.append(["pageviews_proc", "missing values imputation / data deletion",
                      """"Respondent or page identifiers for some pageview entries were
manually corrected using the code below.
Others were deleted and are logged. Check the has_events column to see if any other paradata
events were associated with deleted pageviews.
Code:
pageviews_proc["respondent_id"] = np.where(pageviews_proc["pageview_id"] == 12376, 30754752,
pageviews_proc["respondent_id"])
pageviews_proc["recnum"] = np.where(pageviews_proc["pageview_id"] == 12376, 154,
pageviews_proc["recnum"]) """,
                      missing_ids])
del missing_ids

# Remove temporary variables used for imputation.
pageviews_proc.drop(columns=["page_id_meta", "page_id_merge", "imp_respondent_id",
"imp_recnum", "imp_page_id", "match", "match_load_time_diff",

```

```

        "source_pageview_id"], inplace=True)

# %% Device information from UAS

# Check for missing UAS.
if pageviews_proc["uas"].isna().any():
    processing_log.append(["pageviews_proc", "missing values", "Some pageview entries have
missing UAS",
        pageviews_proc[pageviews_proc["uas"].isna()]])

# Parse UAS to get device info.
pageviews_proc["parsed_uas"] = pageviews_proc["uas"].apply(lambda uas: ua.parse(uas))
pageviews_proc["browser_family"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.browser.family)
pageviews_proc["browser_version"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.browser.version_string)
pageviews_proc["os_family"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.os.family)
pageviews_proc["os_version"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.os.version_string)
pageviews_proc["device_brand"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.device.brand)
pageviews_proc["device_model"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.device.model)
pageviews_proc["is_mobile"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.is_mobile)
pageviews_proc["is_tablet"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.is_tablet)
pageviews_proc["is_pc"] = pageviews_proc["parsed_uas"].apply(lambda i_uas: i_uas.is_pc)
pageviews_proc["touch_capable"] = pageviews_proc["parsed_uas"].apply(lambda i_uas:
i_uas.is_touch_capable)
pageviews_proc["bot"] = pageviews_proc["parsed_uas"].apply(lambda i_uas: i_uas.is_bot)
pageviews_proc.drop("parsed_uas", axis=1, inplace=True)

# Correct known cases of incorrect or duplicate device type. E.g. with some tablets, the UAS
parser identifies more
# than one device type from the UAS. External file - if exists - is used to perform
corrections.
ref_data["uas_corrections"] = pd.read_csv(settings["uas_corrections_file"], quotechar='"')
uas_corrections = ref_data["uas_corrections"]

# Merge by full uas, full model name and partial model name. For the latter two options, we
first need to
# check whether full or partial model name is contained in the entry string and store it to be
used as
# merging variable. Corrections are preserved in this order, i.e. full uas match has the
highest priority while
# partial model match has the lowest priority.
full_model = uas_corrections[uas_corrections["match_type"] ==
"full_model_code"]["search_string"]
partial_model = uas_corrections[uas_corrections["match_type"] ==
"partial_model_code"]["search_string"]
pageviews_proc["full_model_match"] = pageviews_proc["uas"].str.extract("(" +
"|".join(full_model) + ")",
                                                                    expand=False)
pageviews_proc["partial_model_match"] = pageviews_proc["uas"].str.extract(
    "(" + "|".join(partial_model) + ")",
    expand=False)

pageviews_proc = pd.merge(pageviews_proc, uas_corrections, how="left", left_on="uas",
right_on="search_string",
                        indicator="uas_merge")
pageviews_proc = pd.merge(pageviews_proc, uas_corrections, how="left",
left_on="full_model_match",
                        right_on="search_string", suffixes=("", "_1"),
indicator="full_model_merge")
pageviews_proc = pd.merge(pageviews_proc, uas_corrections, how="left",
left_on="partial_model_match",
                        right_on="search_string", suffixes=("", "_2"),
indicator="partial_model_merge")

pageviews_proc[["device_brand_old", "device_model_old", "is_mobile_old", "is_tablet_old",
"is_pc_old"]] = pageviews_proc[["device_brand", "device_model", "is_mobile",
"is_tablet", "is_pc"]]

```

```

# TODO: Putting _corr variables together first may not be necessary and could be directly used
# to replace original
# values.
pageviews_proc["device_brand_corr"] = np.where(
    (pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] == "both"),
    pageviews_proc["device_brand_corr_1"],
    np.where((pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] !=
"both") & (
        pageviews_proc["partial_model_merge"] == "both"),
        pageviews_proc["device_brand_corr_2"],
        pageviews_proc["device_brand_corr"])))
pageviews_proc["device_model_corr"] = np.where(
    (pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] == "both"),
    pageviews_proc["device_model_corr_1"],
    np.where((pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] !=
"both") & (
        pageviews_proc["partial_model_merge"] == "both"),
        pageviews_proc["device_model_corr_2"],
        pageviews_proc["device_model_corr"])))
pageviews_proc["is_mobile_corr"] = np.where(
    (pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] == "both"),
    pageviews_proc["is_mobile_corr_1"],
    np.where((pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] !=
"both") & (
        pageviews_proc["partial_model_merge"] == "both"),
        pageviews_proc["is_mobile_corr_2"],
        pageviews_proc["is_mobile_corr"])))
pageviews_proc["is_tablet_corr"] = np.where(
    (pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] == "both"),
    pageviews_proc["is_tablet_corr_1"],
    np.where((pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] !=
"both") & (
        pageviews_proc["partial_model_merge"] == "both"),
        pageviews_proc["is_tablet_corr_2"],
        pageviews_proc["is_tablet_corr"])))
pageviews_proc["is_pc_corr"] = np.where(
    (pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] == "both"),
    pageviews_proc["is_pc_corr_1"],
    np.where((pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] !=
"both") & (
        pageviews_proc["partial_model_merge"] == "both"),
        pageviews_proc["is_pc_corr_2"],
        pageviews_proc["is_pc_corr"])))
pageviews_proc["match_type"] = np.where(
    (pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] == "both"),
    pageviews_proc["match_type_1"],
    np.where((pageviews_proc["uas_merge"] != "both") & (pageviews_proc["full_model_merge"] !=
"both") & (
        pageviews_proc["partial_model_merge"] == "both"),
        pageviews_proc["match_type_2"],
        pageviews_proc["match_type"])))

pageviews_proc.drop(columns=["device_brand_corr_1", "device_brand_corr_2",
"device_model_corr_1",
                                "device_model_corr_2", "is_mobile_corr_1", "is_mobile_corr_2",
"is_tablet_corr_1",
                                "is_tablet_corr_2", "is_pc_corr_1", "is_pc_corr_2",
"match_type_1", "match_type_2",
                                "search_string", "search_string_1", "search_string_2",
"full_model_match",
                                "partial_model_match", "uas_merge", "full_model_merge",
"partial_model_merge"],
                    inplace=True)

pageviews_proc["device_brand"] = np.where(
    (pageviews_proc["device_brand_corr"].notna()) & (pageviews_proc["match_type"].notna()),
    pageviews_proc["device_brand_corr"],
    pageviews_proc["device_brand"])
pageviews_proc["device_model"] = np.where(
    (pageviews_proc["device_model_corr"].notna()) & (pageviews_proc["match_type"].notna()),
    pageviews_proc["device_model_corr"],
    pageviews_proc["device_model"])
pageviews_proc["is_mobile"] = np.where(
    (pageviews_proc["is_mobile_corr"].notna()) & (pageviews_proc["match_type"].notna()),
    pageviews_proc["is_mobile_corr"],
    pageviews_proc["is_mobile"])

```

```

pageviews_proc["is_tablet"] = np.where(
    (pageviews_proc["is_tablet_corr"].notna()) & (pageviews_proc["match_type"].notna()),
    pageviews_proc["is_tablet_corr"],
    pageviews_proc["is_tablet"])
pageviews_proc["is_pc"] = np.where((pageviews_proc["is_pc_corr"].notna()) &
    (pageviews_proc["match_type"].notna()),
    pageviews_proc["is_pc_corr"],
    pageviews_proc["is_pc"])

# Log changes.
processing_log.append(["pageviews_proc", "data modification",
    "Changes have been applied to some device variables based on the UAS
    corrections file. "
    "Note that the log may also contain some cases for which variables were
    not modified. "
    "This is most likely due to patial model match not bringing additional
    information "
    "to currently available",
    pageviews_proc.loc[pageviews_proc["match_type"].notna(),
        ["pageview_id", "recnum", "device_brand",
        "device_brand_old",
        "device_model", "device_model_old", "is_mobile",
        "is_mobile_old",
        "is_tablet", "is_tablet_old", "is_pc", "is_pc_old",
        "match_type"]]])

# Drop correction variables as they are not longer needed.
pageviews_proc.drop(columns=["device_brand_corr", "device_model_corr", "is_tablet_corr",
    "is_mobile_corr", "is_pc_corr", "match_type", "device_brand_old",
    "device_model_old", "is_mobile_old", "is_tablet_old",
    "is_pc_old"], inplace=True)

# Set nonsubstantive values to missing and change None to NaN for consistency.
pageviews_proc.loc[pageviews_proc["browser_family"] == "", "browser_family"] = np.nan
pageviews_proc.loc[pageviews_proc["browser_version"] == "", "browser_version"] = np.nan
pageviews_proc.loc[pageviews_proc["os_family"] == "", "os_family"] = np.nan
pageviews_proc.loc[pageviews_proc["os_version"] == "", "os_version"] = np.nan
pageviews_proc.loc[(pageviews_proc["device_brand"] == "") | (
    pageviews_proc["device_brand"] == "Generic") | (pageviews_proc["device_brand"] ==
    "Generic_Android"),
    "device_brand"] = np.nan
pageviews_proc.loc[(pageviews_proc["device_model"] == "") | (pageviews_proc["device_model"] ==
    "Smartphone"),
    "device_model"] = np.nan

pageviews_proc["browser_family"].fillna(value=np.nan, inplace=True)
pageviews_proc["browser_version"].fillna(value=np.nan, inplace=True)
pageviews_proc["os_family"].fillna(value=np.nan, inplace=True)
pageviews_proc["os_version"].fillna(value=np.nan, inplace=True)
pageviews_proc["device_brand"].fillna(value=np.nan, inplace=True)
pageviews_proc["device_model"].fillna(value=np.nan, inplace=True)

# Change relevant variables to categorical.
pageviews_proc["browser_family"] = pageviews_proc["browser_family"].astype("category")
pageviews_proc["os_family"] = pageviews_proc["os_family"].astype("category")
pageviews_proc["device_brand"] = pageviews_proc["device_brand"].astype("category")

# Create a unified device type variable and check for potential entries with two or more
devices identified.
pageviews_proc.loc[pageviews_proc["is_pc"] == True, "device_type"] = "computer"
pageviews_proc.loc[pageviews_proc["is_mobile"] == True, "device_type"] = "mobile_phone"
pageviews_proc.loc[pageviews_proc["is_tablet"] == True, "device_type"] = "tablet"
pageviews_proc["device_type"] = pageviews_proc["device_type"].astype("category")

# Check for missing values in the device variable.
if pageviews_proc["device_type"].isna().any():
    processing_log.append(["pageviews_proc", "missing/invalid data",
        "Device type could not be determined from the UAS data for some
        entries.",
        pageviews_proc[pageviews_proc["device_type"].isna()]])

# Check for cases with more than one type of device assigned and set them to missing.
if (pageviews_proc[["is_mobile", "is_tablet", "is_pc"]].sum(axis=1) > 1).any():
    pageviews_proc.loc[pageviews_proc[["is_mobile", "is_tablet", "is_pc"]].sum(axis=1) > 1,
    "device_type"] = np.nan
    processing_log.append(["pageviews_proc", "missing/invalid data",

```

```

        "The UAS parser assigned more than one device type to some pageview
entries. These values were set as missing/invalid.",
        pageviews_proc[pageviews_proc[["is_mobile", "is_tablet",
"is_pc"]].sum(axis=1) > 1]])

# TODO: Preverjanje, če naprava ni konstantna znotraj vsakega respondent_id, pod pogojem, da
je to zahtevano v nastavitvi.
# Treba je upoštevati, da se UAS lahko spremeni zaradi vmesne nadgradnje brskalnika, tako da
morda je za začetek
# OK gledeati, če je tip brskalnika in tip naprave enak. Pri tem kot dodatnih pogoji
sprogramiramo.

# Drop dummy device variables.
pageviews_proc.drop(columns=["is_pc", "is_tablet", "is_mobile"], inplace=True)

del full_model
del partial_model

# %% Final editing and tidying up

# Some pageviews entries occur due to technical reasons that are not completely understood
yet. Sequential pageview entries for the same
# page that have no other recorded paradata events are therefore removed. Because we only
observe pageloads, we do not have to alter
# the timestamp for the kept events under assumption that "ghost" events always occur after
the "valid" event and not before it. The
# check is performed that such pageview is immediately preceded by a valid pageview event on
the same page.
# TODO: Ghost events should be further researched to confirm that they only occur sequentially
after the real event and never before it.
pageviews_proc["new_page"] =
pageviews_proc.sort_values(['respondent_id', 'load_ts']).groupby('respondent_id')['page_id'].diff().astype(bool)
pageviews_proc["has_events"] =
pageviews_proc["pageview_id"].isin(event_list_raw.loc[event_list_raw["event_type"] !=
"pageview", "pageview_id"])

# Pageview duration is calculated for control.
pageviews_proc["pageview_duration"] = pageviews_proc.sort_values(
    ['respondent_id', 'load_ts']).groupby('respondent_id')['load_ts'].diff(-
1).dt.total_seconds()*(-1)

processing_log.append(["pageviews_proc", "data deletion", "Pageview entries without other
paradata events that are adjacent to "
    "another pageview on the same page will be deleted.",
    pageviews_proc[pageviews_proc["has_events"] == False] &
(pageviews_proc["new_page"] == False)])

pageviews_proc = pageviews_proc[(pageviews_proc["has_events"] != False) |
(pageviews_proc["new_page"] != False)]

# Change identifiers to integers.
pageviews_proc["respondent_id"] = pageviews_proc["respondent_id"].astype(int)
pageviews_proc["page_id"] = pageviews_proc["page_id"].astype(int)
pageviews_proc["pageview_id"] = pageviews_proc["pageview_id"].astype(int)
pageviews_proc["recnum"] = pageviews_proc["recnum"].astype(int)

# Keep only relevant variables.
pageviews_proc = pageviews_proc[["pageview_id", "survey_id", "page_id", "respondent_id",
"recnum", "load_ts", "post_ts", "survey_lang",
    "uas", "browser_family", "browser_version", "os_family",
"os_version", "device_brand", "device_model",
    "touch_capable", "bot", "device_type", "pixel_ratio",
"screen_width", "screen_height", "screen_width_avail",
    "screen_height_avail", "viewport_width", "viewport_height",
"page_width", "page_height"]]

# %% RESPONSES PROCESSING
responses_proc = responses_raw.copy()

# Merge responses with page id and respondent id info.
responses_proc = pd.merge(responses_proc, pageviews_proc[["pageview_id", "page_id",
"respondent_id"]],
    on="pageview_id", how="left", indicator="pageviews_merge")

if (responses_proc["pageviews_merge"] != "both").any():

```



```

        processing_log.append(["responses_proc", "merging failure",
                               "Some response entries were not matched with pageviews
information.",
                               responses_proc[responses_proc["pageviews_merge"] != "both"]])

# TODO: Question order information from metadata.

# Drop missing identifiers
# TODO: CHECK AND LOG!!!!!!!!!!!!!!
responses_proc = responses_proc[responses_proc["respondent_id"].notna()]

# Change identifiers to integers.
responses_proc["response_id"] = responses_proc["response_id"].astype(int)
responses_proc["pageview_id"] = responses_proc["pageview_id"].astype(int)
responses_proc["respondent_id"] = responses_proc["respondent_id"].astype(int)
responses_proc["page_id"] = responses_proc["page_id"].astype(int)
responses_proc["question_id"] = responses_proc["question_id"].astype(int)

# TODO NEKO IME, ČE NI ITEM JE QUESTION a je sploh treba
responses_proc["item_name"] = np.where(responses_proc["item_name"].isna(),
responses_proc["question_name"], responses_proc["item_name"] )

# Keep only relevant variables.
responses_proc = responses_proc[["response_id", "pageview_id", "respondent_id", "page_id",
"response_ts", "question_id", "question_name",
                                "question_type", "question_input_type", "item_id",
"item_name", "response_option_id",
                                "response_option_unique_id", "response_option_code",
"response_action", "response_value"]]

# %% FOCUS CHANGES
focus_changes_proc = focus_changes_raw.copy()

# Add respondent and page ids.
focus_changes_proc = pd.merge(focus_changes_proc, pageviews_proc[["pageview_id",
"respondent_id", "page_id"]],
                                on="pageview_id", how="left", indicator="pageviews_merge")
if (focus_changes_proc["pageviews_merge"] != "both").any():
    processing_log.append(["focus_changes_proc", "merging failure",
                           "Some focus change entries were not matched with pageviews.",
                           focus_changes_proc[focus_changes_proc["pageviews_merge"] !=
"both"]])

# %% Study characteristics of events

# Time between page load and focus change event.
focus_changes_proc.sort_values(["respondent_id", "pageview_id", "focus_change_ts"],
inplace=True)
focus_changes_proc = pd.merge(focus_changes_proc, pageviews_proc[["pageview_id", "load_ts",
"browser_family", "browser_version",
                                                                    "os_family", "os_version",
"device_brand", "device_model"]],
                                on="pageview_id", how="left")
focus_changes_proc["load_time_diff"] = (focus_changes_proc["focus_change_ts"] -
focus_changes_proc["load_ts"]).dt.total_seconds()

# Number of focus and blur events by pageviews.
# This code can be run also later to check the effects of applied changes.
event_type_counts = focus_changes_proc.groupby(["pageview_id",
"change_type"]).agg(n_events=("focus_change_id", "count")).reset_index()
event_type_counts = event_type_counts.pivot(index="pageview_id", columns="change_type")
event_type_counts.columns = ['_'.join(col).strip() for col in
event_type_counts.columns.values]
event_type_counts.reset_index(inplace=True)
pageviews_focus_events = pd.merge(pageviews_proc, event_type_counts, on="pageview_id",
how="left")
pageviews_focus_events["n_events_blur"].fillna(0, inplace=True)
pageviews_focus_events["n_events_focus"].fillna(0, inplace=True)
pageviews_focus_events["n_events_match"] = np.where((pageviews_focus_events["n_events_blur"] -
pageviews_focus_events["n_events_focus"]) == 0,
True, False)
pageviews_focus_events = pageviews_focus_events[["pageview_id", "respondent_id", "page_id",
"load_ts",
                                                'n_events_blur', 'n_events_focus',
"n_events_match", "uas", "browser_family", "browser_version", "os_family", "os_version"]]

```

```

pageviews_focus_events.sort_values(['respondent_id', 'load_ts'], inplace=True)
del event_type_counts

# %% Remove identified "invalid" events
# It is not always clear which events are valid in terms of truly indicating that the
respondent left the survey window/tab. Only most
#

# Identify and remove events that (all deleted events are logged):
# - are recorded as occurred earlier than the corresponding page was loaded;
focus_changes_proc["flag_neg_time"] = np.where(focus_changes_proc["load_time_diff"] < 0, True,
False)
del_focus_events = focus_changes_proc[focus_changes_proc["flag_neg_time"] == True]
focus_changes_proc = focus_changes_proc[focus_changes_proc["flag_neg_time"] == False]

# - are focus events occurring less than 0.1 seconds after the page was loaded (some
browsers appear to trigger focus upon pageload);
focus_changes_proc["flag_focus_lt_100ms"] = np.where((focus_changes_proc["load_time_diff"] <
0.1) & (focus_changes_proc["change_type"] == "focus"),
True, False)
del_focus_events =
del_focus_events.append(focus_changes_proc[focus_changes_proc["flag_focus_lt_100ms"] == True])
focus_changes_proc = focus_changes_proc[focus_changes_proc["flag_focus_lt_100ms"] == False]

# - are repeated (i.e. two focus or blur events in row) events less than 0.1 second after
the same type of event;
focus_changes_proc.sort_values(["respondent_id", "pageview_id", "focus_change_ts"],
inplace=True)
focus_changes_proc["flag_repeated"] =
focus_changes_proc["change_type"].eq(focus_changes_proc["change_type"].shift(1))
focus_changes_proc.loc[focus_changes_proc["pageview_id"] !=
focus_changes_proc["pageview_id"].shift(1), "flag_repeated"] = False
focus_changes_proc["time_diff_repeated"] = focus_changes_proc.sort_values(
["respondent_id", "pageview_id",
"focus_change_ts"]).groupby('pageview_id')['focus_change_ts'].diff().dt.total_seconds()
focus_changes_proc["time_diff_repeated"] = np.where(focus_changes_proc["flag_repeated"] ==
True,
focus_changes_proc["time_diff_repeated"],
pd.NaT)
focus_changes_proc["flag_repeated_100ms"] = (focus_changes_proc["flag_repeated"] == True) &
(focus_changes_proc["time_diff_repeated"] < 0.1)
del_focus_events =
del_focus_events.append(focus_changes_proc[focus_changes_proc["flag_repeated_100ms"] == True])
focus_changes_proc = focus_changes_proc[focus_changes_proc["flag_repeated_100ms"] == False]

processing_log.append(["focus_changes_proc", "data deletion",
""
Focus change entries matching the following criteria were deleted:
- were recorded as occurred earlier than the corresponding page was loaded;
- were focus events occurring less than 0.1 seconds after the page was loaded;
- were repeated events less than 0.1 second after the same type of event;
"", del_focus_events])
del del_focus_events

# Drop missing identifiers
# TODO: CHECK AND LOG!!!!!!!!!!!!!!
focus_changes_proc = focus_changes_proc[focus_changes_proc["respondent_id"].notna()]

# Change identifiers to integers.
focus_changes_proc["focus_change_id"] = focus_changes_proc["focus_change_id"].astype(int)
focus_changes_proc["pageview_id"] = focus_changes_proc["pageview_id"].astype(int)
focus_changes_proc["respondent_id"] = focus_changes_proc["respondent_id"].astype(int)
focus_changes_proc["page_id"] = focus_changes_proc["page_id"].astype(int)

# Reorder and keep selected variables.
focus_changes_proc = focus_changes_proc[["focus_change_id", "pageview_id", "respondent_id",
"page_id", "focus_change_ts", "change_type"]]

# %% MESSAGES
messages_proc = messages_raw.copy()

# Add respondent and page ids.
messages_proc = pd.merge(messages_proc, pageviews_proc[["pageview_id", "respondent_id",
"page_id"]],
on="pageview_id", how="left", indicator="pageviews_merge")
if (messages_proc["pageviews_merge"] != "both").any():

```

```

processing_log.append(["messages_proc", "merging failure",
                      "Some response message entries were not matched with pageviews.",
                      messages_proc[messages_proc["pageviews_merge"] != "both"]])

# Merge messages with the same display_ts.
# TODO: Check here or during analysis whether any other events should be merged (e.g. due to
# very short time between two messages during
# the same pageview).
messages_proc["msg_group_id"] = messages_proc.groupby(["pageview_id", "msg_type",
"display_ts"]).grouper.group_info[0]
processing_log.append(["messages_proc", "note", "data aggregation",
                      "Message display entries with the same display_ts will be merged into a
single entry. Message IDs will be replaced. "
                      "The log contains a copy of full dataframe before this change is
applied.", messages_proc])
messages_proc["msg_id"] = messages_proc["msg_group_id"]
# We only list all trigger question names, not IDs, because question-level information is
pretty useless anyway. In the future, item-level
# info may be added.
messages_proc["trigger_question_names"] =
messages_proc.groupby("msg_id")["trigger_question_name"].transform(lambda x:
', '.join(x.astype(str)))
messages_proc.drop(columns=["trigger_question_id", "trigger_question_name", "msg_group_id"],
inplace=True)
# Only the record with the latest timestamp is kept.
messages_proc = messages_proc.sort_values(["msg_id",
"close_ts"]).groupby("msg_id").last().reset_index()

# Drop missing identifiers
# TODO: CHECK AND LOG!!!!!!!!!!!!!!
messages_proc = messages_proc[messages_proc["respondent_id"].notna()]

# Change identifiers to integers.
messages_proc["msg_id"] = messages_proc["msg_id"].astype(int)
messages_proc["pageview_id"] = messages_proc["pageview_id"].astype(int)
messages_proc["respondent_id"] = messages_proc["respondent_id"].astype(int)
messages_proc["page_id"] = messages_proc["page_id"].astype(int)

# Reorder and keep selected variables.
messages_proc = messages_proc[["msg_id", "pageview_id", "respondent_id", "page_id",
"display_ts", "close_ts", "msg_type",
                                "trigger_question_names", "ignorable", "msg_text",
"respondent_action"]]

# %% POINTER MOVEMENTS
pointer_proc = pointer_raw.copy()

# Calculate Euclidean distance to confirm zero-distance events.
pointer_proc["euclid_distance"] = np.sqrt((pointer_proc["end_coord_x"] -
pointer_proc["start_coord_x"])**2 + (
                                pointer_proc["end_coord_y"] -
pointer_proc["start_coord_y"])**2)

# Drop zero-distance events, but only if both Euclidean distance and move_distance are zero.
processing_log.append(["pointer_proc", "data deletion",
                      "Zero-distance pointer movement events will be removed if recorded andy
Euclidean distance are both zero.",
                      pointer_proc[(pointer_proc["move_distance"] == 0) &
(pointer_proc["euclid_distance"] == 0)])
pointer_proc = pointer_proc[(pointer_proc["move_distance"] != 0) |
(pointer_proc["euclid_distance"] != 0)]

# Problematic are only events where Euclidean distance is not zero and move_distance is zero.
It is possible - yet odd - that the Euclidean
# distance is zero if the movement starts and ends at the same point.
if ((pointer_proc["euclid_distance"] != 0) & (pointer_proc["move_distance"] == 0)).any():
    processing_log.append(["pointer_proc", "inconsistent data", "Some entries have non-zero
Euclidean distance but zero move distance.",
                          pointer_proc[(pointer_proc["euclid_distance"] != 0) &
(pointer_proc["move_distance"] == 0)])

# Add respondent and page ids.
pointer_proc = pd.merge(pointer_proc, pageviews_proc[["pageview_id", "respondent_id",
"page_id"]], on="pageview_id", how="left",
                        indicator="pageviews_merge")
if (pointer_proc["pageviews_merge"] != "both").any():

```

```

processing_log.append(["pointer_proc", "merging failure",
                      "Some response message entries were not matched with pageviews.",
                      pointer_proc[pointer_proc["pageviews_merge"] != "both"]])

# Drop missing identifiers
# TODO: CHECK AND LOG!!!!!!!!!!!!!!
pointer_proc = pointer_proc[pointer_proc["respondent_id"].notna()]

# Change identifiers to integers.
pointer_proc["move_id"] = pointer_proc["move_id"].astype(int)
pointer_proc["pageview_id"] = pointer_proc["pageview_id"].astype(int)
pointer_proc["respondent_id"] = pointer_proc["respondent_id"].astype(int)
pointer_proc["page_id"] = pointer_proc["page_id"].astype(int)

# Reorder and keep selected variables.
pointer_proc = pointer_proc[["move_id", "pageview_id", "respondent_id", "page_id",
                             "move_start_ts", "move_end_ts",
                             "start_coord_x", "start_coord_y", "end_coord_x", "end_coord_y",
                             "move_distance"]]

# %% CLICKS
clicks_proc = clicks_raw.copy()

# Many click events seem to be recorded more than once. Identify clicks in which coordinate
change compared to the previous click.
clicks_proc.sort_values(["pageview_id", "click_ts"], inplace=True)
clicks_proc["diff_coord"] = np.where((clicks_proc["coord_x"].diff() == 0) &
                                     (clicks_proc["coord_y"].diff() == 0), False, True)
clicks_proc.loc[clicks_proc["pageview_id"] != clicks_proc["pageview_id"].shift(1),
               "diff_coord"] = True

# Time difference between events is used as further criterion to reduce errors in separating
click events. This may be improved in the
# future by analysing whether a group of click events refers to the same or different UI
elements.
# TODO: Recheck the logic behind merging events.
clicks_proc["same_coord_time_diff"] = clicks_proc.sort_values(["pageview_id",
                                                             "click_ts"])["click_ts"].diff().dt.total_seconds()
clicks_proc.loc[clicks_proc["pageview_id"] != clicks_proc["pageview_id"].shift(1),
               "same_coord_time_diff"] = np.NaN
clicks_proc["base_event"] = np.where((clicks_proc["diff_coord"] == False) &
                                     (clicks_proc["same_coord_time_diff"] < 0.1), False, True)

# Give the same identifier to all sequential clicks designated for merging. This is done by
removing identifiers for such events
# then forward filling the ID of the first click event.
clicks_proc["click_id"] = clicks_proc["click_id"] * clicks_proc["base_event"]
clicks_proc["click_id"].replace(0, np.nan, inplace=True)
clicks_proc["click_id"].fillna(method="ffill", inplace=True)

# TODO: Currently we just remove all non-base events without even logging themn.
# This needs to be changed to pivoting that will enable analysis of event types.
clicks_proc = clicks_proc[clicks_proc["base_event"] == True]

# # Define new IDs based on groups of events
# clicks_proc["click_group_id"] = np.where(clicks_proc["same_coord"] == True,
# clicks_proc["click_id"].shift(1), clicks_proc["click_id"])
# clicks_proc["click_group_id"] = clicks_proc["click_group_id"].rank(method="dense")

# # Sequentially count the number of click events in each group (the number is increased by
one to get the total number of events in group).
# clicks_proc["count"] = clicks_proc.sort_values(["pageview_id",
# "click_ts"]).groupby("click_group_id")["click_group_id"].cumcount()
# clicks_proc["count"] += 1
# # More than two events grouped together is unexpected and should be handled. Errors will not
neccessarily occur, but it should be checked
# # anyway.
# if clicks_proc["count"].max() > 2:
#     processing_log.append(["clicks_proc", "unexpected data",
#                             "Some groups of click events contain more than two entries, wich
can cause unexpected results. Adapt procedures if needed",
#                             clicks_proc[clicks_proc["count"].max() > 2]])

# # Merge click events ccuring next to each other with the same coordinates. The earliest
timestamp and all div-related info is retained.

```

```

# processing_log.append(["clicks_proc", "note", "data aggregation",
# "Click events occurring next to each other with the same coordinates
# will be merged into a single entry. "
# "Click IDs will be replaced. The log contains a copy of full
# dataframe before this change is applied.",
# clicks_proc])

# # Use the first event timestamp and replace original click IDs with group IDs.
# clicks_proc["click_ts"] = clicks_proc.sort_values(["pageview_id",
# "click_ts"]).groupby("click_group_id")["click_ts"].transform("first")
# clicks_proc["click_id"] = clicks_proc["click_group_id"]

# grouped_clicks = clicks_proc.pivot(index="click_id",
# columns="count",
# values=["div_id", "div_class", "div_type", "same_coord"])

# Add respondent and page ids.
clicks_proc = pd.merge(clicks_proc, pageviews_proc[["pageview_id", "respondent_id",
"page_id"]], on="pageview_id", how="left",
indicator="pageviews_merge")
if (clicks_proc["pageviews_merge"] != "both").any():
    processing_log.append(["clicks_proc", "merging failure",
        "Some click entries were not matched with pageviews.",
        clicks_proc[clicks_proc["pageviews_merge"] != "both"]])

# Drop missing identifiers
# TODO: CHECK AND LOG!!!!!!!!!!!!!!
clicks_proc = clicks_proc[clicks_proc["respondent_id"].notna()]

# Change identifiers to integers.
clicks_proc["click_id"] = clicks_proc["click_id"].astype(int)
clicks_proc["pageview_id"] = clicks_proc["pageview_id"].astype(int)
clicks_proc["respondent_id"] = clicks_proc["respondent_id"].astype(int)
clicks_proc["page_id"] = clicks_proc["page_id"].astype(int)

# Reorder and keep selected variables.
clicks_proc = clicks_proc[["click_id", "pageview_id", "respondent_id", "page_id", "click_ts",
"coord_x", "coord_y"]]

%% VIEW CHANGES
# TODO: This for now only includes orientation changes, everything else is dropped.
view_changes_proc = view_changes_raw[view_changes_raw["change_type"] == "orientation_change"]

# Add respondent and page ids.
view_changes_proc = pd.merge(view_changes_proc, pageviews_proc[["pageview_id",
"respondent_id", "page_id"]], on="pageview_id", how="left",
indicator="pageviews_merge")
if (view_changes_proc["pageviews_merge"] != "both").any():
    processing_log.append(["view_changes_proc", "merging failure",
        "Some view changes entries were not matched with pageviews.",
        view_changes_proc[view_changes_proc["pageviews_merge"] != "both"]])

# %% EVENTS LIST
# Processed events list excludes events that were excluded during processing of individual
# datasets.

# - pointer_move_start and pointer_move_end (between the two events is active period that
# should be considered).

```

2.4 Basic analysis

TODO: Imena pomoćnih objekata za agregaciju naj bodo konsistentna. Glej npr. za pointer kot primer.

%% SETUP

```

settings["results_folder"] = "/Users/nejc/WD/Python/Paradata/"
settings["use_common_names"] = True
settings["export_to_excel"] = False

```

%% SOME HELPER STUFF

Matching recnums and respondent ids.

```

ref_recnum_id = pageviews_proc.groupby(["respondent_id", "recnum"]).size().reset_index()
ref_recnum_id.drop(columns=0, inplace=True)

# Matching item IDs and names.
ref_item_names = responses_proc.groupby(["page_id", "item_id",
"item_name"]).size().reset_index()
ref_item_names = pd.merge(ref_item_names, survey_metadata["pages"][["page_id", "page_seq"]],
on="page_id", how="left")
ref_item_names.drop(columns=0, inplace=True)

# Matching question IDs and names.
ref_question_names = responses_proc.groupby(["question_id",
"question_name"]).size().reset_index()
ref_question_names.drop(columns=0, inplace=True)

# Rules for common item names (i.e. due to having experimental versions of items).
ref_item_versions = pd.read_csv(settings["item_versions_file"])
ref_common_item_names = pd.melt(ref_item_versions, id_vars=["common_name"],
value_vars=["version_1", "version_2", "version_3",
"version_4", "version_5"], value_name="item name")
ref_common_item_names.drop(columns="variable", inplace=True)

# %% FOCUS OUTS
focus_out_analysis = focus_changes_proc.copy()

#
focus_out_analysis.sort_values(["respondent_id", "pageview_id", "focus_change_id"],
inplace=True)

# For sequentially equal events during the same pageview we take the latest event for blur and
the earliest event for focus.
# We do this by setting equal ID to all sequentially equal events, then taking the first or
the last event in the group depending on
# the type.
focus_out_analysis["seq_equal"] = np.where((focus_out_analysis["change_type"] ==
focus_out_analysis["change_type"].shift(1)) & (focus_out_analysis["pageview_id"] ==
focus_out_analysis["pageview_id"].shift(1)), True, False)
focus_out_analysis["focus_change_id"] = np.where(focus_out_analysis["seq_equal"] == True,
np.nan, focus_out_analysis["focus_change_id"])
focus_out_analysis["focus_change_id"].fillna(method="ffill", inplace=True)
focus_out_analysis["seq_min_ts"] =
focus_out_analysis.groupby("focus_change_id")["focus_change_ts"].transform("min")
focus_out_analysis["seq_max_ts"] =
focus_out_analysis.groupby("focus_change_id")["focus_change_ts"].transform("max")
focus_out_analysis = focus_out_analysis[((focus_out_analysis["change_type"] == "focus") &
(focus_out_analysis["focus_change_ts"] ==
focus_out_analysis["seq_min_ts"])) |
((focus_out_analysis["change_type"] == "blur") &
(focus_out_analysis["focus_change_ts"] ==
focus_out_analysis["seq_max_ts"]))]

# Duration of focus out event. Only relevant are durations calculated from previous blur to
the next focus. The code above eliminates all
# sequential repeated events on the same page, therefore focus event will be preceded on the
page by blur event. However, if focus event is
# the first on the page, then duration is not valid and will be missing.
focus_out_analysis["duration"] =
focus_out_analysis["focus_change_ts"].diff().dt.total_seconds()
focus_out_analysis["focus_out_ts"] = focus_out_analysis["focus_change_ts"].shift(1)
focus_out_analysis["duration"] = np.where((focus_out_analysis["change_type"] == "blur") |
(focus_out_analysis["pageview_id"] !=
focus_out_analysis["pageview_id"].shift(1)),
np.nan, focus_out_analysis["duration"])

# TODO: Preveri od kod negativna trajanja, tule jih damo samo na nič. Tule tudi odstranimo
primerke na zadnji strani,
# ker drugače nagajajo adjusted časi. Premisli ponovno!
focus_out_analysis["duration"] = np.where(focus_out_analysis["duration"] < 0, 0,
focus_out_analysis["duration"])
focus_out_analysis = focus_out_analysis[focus_out_analysis["page_id"] != -2]

# Keep only focus events with valid duration.

```

```

# TODO: Log deleted entries.
focus_out_analysis = focus_out_analysis[focus_out_analysis["duration"].notna()]
focus_out_analysis["duration_geq_5s"] = np.where(focus_out_analysis["duration"] >= 5, True,
False)

focus_out_analysis.rename(columns={"focus_change_ts": "focus_in_ts"}, inplace=True)
focus_out_analysis = focus_out_analysis[["focus_change_id", "pageview_id", "respondent_id",
"page_id", "focus_in_ts",
"focus_out_ts", "duration", "duration_geq_5s"]]

# Aggregate focus events at the respondent level and calculate adjusted pageview times. This
is used later to calculate adjusted pageview
# times.
respondent_focus_pageview = focus_out_analysis.groupby(["respondent_id",
"pageview_id"]).agg(n_focus_out=("focus_change_id", "count"),

n_focus_out_5s=("duration_geq_5s", "sum"),

focus_out_duration=("duration", "sum")).reset_index()

# Aggregation at the page level and calculation of adjusted page
respondent_focus_page = focus_out_analysis.groupby(["respondent_id",
"page_id"]).agg(n_focus_out=("focus_change_id", "count"),

n_focus_out_5s=("duration_geq_5s", "sum"),

focus_out_duration=("duration", "sum")).reset_index()

# Aggregation at the respondent level.
respondent_focus_out_sum =
focus_out_analysis.groupby("respondent_id").agg(n_focus_out=("focus_change_id", "count"),

n_focus_out_5s=("duration_geq_5s", "sum"),

focus_out_duration=("duration", "sum")).reset_index()

# Add page order.
respondent_focus_page = pd.merge(respondent_focus_page, survey_metadata["pages"][["page_id",
"page_seq"]], on="page_id", how="left")
respondent_focus_page.sort_values("page_seq", inplace=True)

# Results table.
focus_out_results = respondent_focus_page.pivot(index="respondent_id", columns="page_seq",
values=["n_focus_out", "n_focus_out_5s",

"focus_out_duration"])
focus_out_results.columns = [tup[0] + "_p" + str(tup[1]) for tup in
focus_out_results.columns.values]
focus_out_results.reset_index(inplace=True)
focus_out_results = pd.merge(respondent_focus_out_sum, focus_out_results, on="respondent_id")
focus_out_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]], focus_out_results,
on="respondent_id")

# Export to Excel
if settings["export_to_excel"]:
    focus_out_results.to_excel(settings["results_folder"] + "focus_out_respondent_level.xlsx",
index=False)

# %% PAGEVIEWS AND FOCUS OUTS
# We do this together to exchange information between the two tables on the fly. It is a bit
of a mess, though.
pageviews_analysis = pageviews_proc.copy()
focus_out_analysis = focus_changes_proc.copy()

# %% Pageviews duration

# Sort values by respondent and timestamp.
pageviews_analysis.sort_values(['respondent_id', 'load_ts'], inplace=True)

# Pageviews duration and pageview end estimate (which is simply adding duration to load_ts to
get next pageview load ts to the same row).
pageviews_analysis["pview_dur"] = pageviews_analysis.sort_values(

```

```

        ['respondent_id', 'load_ts']).groupby('respondent_id')['load_ts'].diff(-
1).dt.total_seconds()*(-1)
pageviews_analysis["end_ts"] = pageviews_analysis["load_ts"] +
pd.to_timedelta(pageviews_analysis["pview_dur"], unit="s")

# %% Focus outs
focus_out_analysis = pd.merge(focus_out_analysis, pageviews_analysis[["pageview_id",
"load_ts", "end_ts"]], on="pageview_id", how="left")
focus_out_analysis["event_outside_ts"] = np.where((focus_out_analysis["focus_change_ts"] <
focus_out_analysis["load_ts"]) |
                                                    (focus_out_analysis["focus_change_ts"] >
focus_out_analysis["end_ts"]), True, False)
focus_out_analysis = focus_out_analysis[focus_out_analysis["event_outside_ts"] == False]

# For sequentially equal events during the same pageview we take the latest event for blur and
the earliest event for focus.
# We do this by setting equal ID to all sequentially equal events, then taking the first or
the last event in the group depending on
# the type.
focus_out_analysis.sort_values(["respondent_id", "pageview_id", "focus_change_id"],
inplace=True)
focus_out_analysis["seq_equal"] = np.where((focus_out_analysis["change_type"] ==
focus_out_analysis["change_type"].shift(1)) & (focus_out_analysis["pageview_id"] ==
focus_out_analysis["pageview_id"].shift(1)), True, False)
focus_out_analysis["focus_change_id"] = np.where(focus_out_analysis["seq_equal"] == True,
np.nan, focus_out_analysis["focus_change_id"])
focus_out_analysis["focus_change_id"].fillna(method="ffill", inplace=True)
focus_out_analysis["seq_min_ts"] =
focus_out_analysis.groupby("focus_change_id")["focus_change_ts"].transform("min")
focus_out_analysis["seq_max_ts"] =
focus_out_analysis.groupby("focus_change_id")["focus_change_ts"].transform("max")
focus_out_analysis = focus_out_analysis[((focus_out_analysis["change_type"] == "focus") &
(focus_out_analysis["focus_change_ts"] ==
focus_out_analysis["seq_min_ts"])) |
                                         ((focus_out_analysis["change_type"] == "blur") &
(focus_out_analysis["focus_change_ts"] ==
focus_out_analysis["seq_max_ts"]))]

# Duration of focus out event. Only relevant are durations calculated from previous blur to
the next focus. The code above eliminates all
# sequential repeated events on the same page, therefore focus event will be preceded on the
page by blur event. However, if focus event is
# the first on the page, then duration is not valid and will be missing.
focus_out_analysis["duration"] =
focus_out_analysis["focus_change_ts"].diff().dt.total_seconds()
focus_out_analysis["focus_out_ts"] = focus_out_analysis["focus_change_ts"].shift(1)
focus_out_analysis["duration"] = np.where((focus_out_analysis["change_type"] == "blur") |
(focus_out_analysis["pageview_id"] !=
focus_out_analysis["pageview_id"].shift(1)),
np.nan, focus_out_analysis["duration"])

# TODO: Preveri od kod negativna trajanja, tule jih damo samo na nič. Tule tudi odstranimo
primerke na zadnji strani,
# ker drugače nagajajo adjusted časi. Premisli ponovno!
focus_out_analysis["duration"] = np.where(focus_out_analysis["duration"] < 0, 0,
focus_out_analysis["duration"])
focus_out_analysis = focus_out_analysis[focus_out_analysis["page_id"] != -2]

# Keep only focus events with valid duration.
# TODO: Log deleted entries.
focus_out_analysis = focus_out_analysis[focus_out_analysis["duration"].notna()]
focus_out_analysis["duration_geq_5s"] = np.where(focus_out_analysis["duration"] >= 5, True,
False)

focus_out_analysis.rename(columns={"focus_change_ts": "focus_in_ts"}, inplace=True)
focus_out_analysis = focus_out_analysis[["focus_change_id", "pageview_id", "respondent_id",
"page_id", "focus_in_ts",
                                         "focus_out_ts", "duration", "duration_geq_5s"]]

# Aggregate focus events at the respondent level and calculate adjusted pageview times. This
is used later to calculate adjusted pageview
# times.
respondent_focus_pageview = focus_out_analysis.groupby(["respondent_id",
"pageview_id"]).agg(n_focus_out=("focus_change_id", "count"),

```



```

n_focus_out_5s=("duration_geq_5s", "sum"),

focus_out_duration=("duration", "sum")).reset_index()

# Aggregation at the page level and calculation of adjusted page
respondent_focus_page = focus_out_analysis.groupby(["respondent_id",
"page_id"]).agg(n_focus_out=("focus_change_id", "count"),

n_focus_out_5s=("duration_geq_5s", "sum"),

focus_out_duration=("duration", "sum")).reset_index()

# Aggregation at the respondent level.
respondent_focus_out_sum =
focus_out_analysis.groupby("respondent_id").agg(n_focus_out=("focus_change_id", "count"),

n_focus_out_5s=("duration_geq_5s", "sum"),

focus_out_duration=("duration", "sum")).reset_index()

# Add page order.
respondent_focus_page = pd.merge(respondent_focus_page, survey_metadata["pages"][["page_id",
"page_seq"]], on="page_id", how="left")
respondent_focus_page.sort_values("page_seq", inplace=True)

# Results table.
focus_out_results = respondent_focus_page.pivot(index="respondent_id", columns="page_seq",
values=["n_focus_out", "n_focus_out_5s",

"focus_out_duration"])
focus_out_results.columns = [tup[0] + "_p" + str(tup[1]) for tup in
focus_out_results.columns.values]
focus_out_results.reset_index(inplace=True)
focus_out_results = pd.merge(respondent_focus_out_sum, focus_out_results, on="respondent_id")
focus_out_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]], focus_out_results,
on="respondent_id")

# Export to Excel
if settings["export_to_excel"]:
    focus_out_results.to_excel(settings["results_folder"] + "focus_out_respondent_level.xlsx",
index=False)

# %% Further processing of pageviews.
# Pageviews duration adjusted for focus out.
pageviews_analysis = pd.merge(pageviews_analysis, respondent_focus_pageview[["pageview_id",
"focus_out_duration"]], on="pageview_id", how="left")
pageviews_analysis["focus_out_duration"].fillna(value=0, inplace=True)
pageviews_analysis["pview_dur_foc"] = pageviews_analysis["pview_dur"] -
pageviews_analysis["focus_out_duration"]

# Number of pageviews and duration of the first and total pageviews. Total pageviews count
includes page visits and pageviews that are
# not the result of page changes (e.g. refreshes, but also potential "ghost" pageviews that
have not been filtered out
# during the data processing).
respondent_pageviews = pageviews_analysis.sort_values(['respondent_id', 'load_ts']).groupby(
["respondent_id", "page_id"]).agg(n_pageviews=("pageview_id", "count"),
duration_1st_pageview=("pview_dur", "first"),
dur_tot=("pview_dur", "sum"),
duration_1st_pageview_foc=("pview_dur_foc", "first"),
dur_tot_foc=("pview_dur_foc", "sum")).reset_index()
respondent_pageviews["dur_tot"] = np.where(respondent_pageviews["dur_tot"] == 0, np.nan,
respondent_pageviews["dur_tot"])
respondent_pageviews["dur_tot_foc"] = np.where(respondent_pageviews["dur_tot_foc"] == 0,
np.nan,
respondent_pageviews["dur_tot_foc"])

# Page visits.
# Check whether the pageview refers to a page that is different to previous (i.e. pageview
event due to moving to a
# different page) and assign same visit_id if not # Type conversion to bool uses the fact that
bool(np.nan) = True.
# TODO: Calculate time of the first VISIT.
pageviews_analysis.sort_values(['respondent_id', 'load_ts'], inplace=True)

```

```

pageviews_analysis["new_visit"] =
pageviews_analysis.groupby('respondent_id')['page_id'].diff().astype(bool)
pageviews_analysis["visit_id"] = pageviews_analysis["pageview_id"] *
pageviews_analysis["new_visit"]
pageviews_analysis["visit_id"].replace(0, np.nan, inplace=True)
pageviews_analysis["visit_id"].fillna(method="ffill", inplace=True)

# Calculate duration of individual visits as the sum of pageviews durations belonging to the
same visit.
visits_analysis = pageviews_analysis[["pageview_id", "visit_id", "page_id", "respondent_id",
"new_visit", "load_ts", "pview_dur",
"pview_dur_foc"]].copy()
visits_analysis["vis_dur"] = visits_analysis.groupby(["visit_id"])["pview_dur"].transform(sum)
visits_analysis["vis_dur_foc"] =
visits_analysis.groupby(["visit_id"])["pview_dur_foc"].transform(sum)
visits_analysis = visits_analysis[visits_analysis["new_visit"] == True]
visits_analysis["vis_dur_foc"] = np.where(visits_analysis["vis_dur_foc"] == 0, np.nan,
visits_analysis["vis_dur_foc"])
visits_analysis["vis_dur"] = np.where(visits_analysis["vis_dur"] == 0, np.nan,
visits_analysis["vis_dur"])
visits_analysis.sort_values(['respondent_id', 'load_ts'], inplace=True)

# Respondent-level statistics of visits.
visits_analysis.sort_values(['respondent_id', 'load_ts'], inplace=True)
respondent_visits = visits_analysis.groupby(["respondent_id",
"page_id"]).agg(n_p_visits=("visit_id", "count"),

dur_vis1=("vis_dur", "first"),

dur_vis1_foc=("vis_dur_foc", "first")).reset_index()
respondent_visits["dur_vis1"] = np.where(respondent_visits["dur_vis1"] == 0, np.nan,
respondent_visits["dur_vis1"])
respondent_visits["dur_vis1_foc"] = np.where(respondent_visits["dur_vis1_foc"] == 0, np.nan,
respondent_visits["dur_vis1_foc"])
respondent_visits["multiple_page_visits"] = np.where(respondent_visits["n_p_visits"] > 1,
True, False)

# Merge with pageviews.
respondent_pageviews = pd.merge(respondent_pageviews, respondent_visits, on=["respondent_id",
"page_id"])

# Device onfo.
# TODO: UAS-based info currently uses the first value. Because there are some instances of
unexplained UAS changes across pageviews,
# more informed decision about which info to keep should be developed.
respondent_device = pageviews_analysis.groupby("respondent_id").agg(
device_type=("device_type", "first"),

browser_family=("browser_family", "first"),

browser_version=("browser_version", "first"),

os_family=("os_family", "first"),

os_version=("os_version", "first"),

device_brand=("device_brand", "first"),

device_model=("device_model", "first"),

touch_capable=("touch_capable", "first")).reset_index()

respondent_pageviews_sum =
respondent_pageviews.groupby("respondent_id").agg(n_pageviews=("n_pageviews", "sum"),
n_p_visits=("n_p_visits", "sum"),
n_visited_pages=("page_id",
"nunique"),
dur_tot=("dur_tot", "sum"),
dur_tot_foc=("dur_tot_foc", "sum"))

respondent_pageviews_sum = pd.merge(respondent_pageviews_sum, respondent_device,
on="respondent_id")

# Number of pages with multiple visits.

```

```

respondent_visits_sum = respondent_visits[respondent_visits["multiple_page_visits"] ==
True].groupby("respondent_id").agg(n_mult_vis_p=("page_id", "nunique")).reset_index()
respondent_pageviews_sum = pd.merge(respondent_pageviews_sum, respondent_visits_sum,
on="respondent_id", how="left")
respondent_pageviews_sum["n_mult_vis_p"] = respondent_pageviews_sum["n_mult_vis_p"].fillna(0)
del respondent_visits_sum

# Add page order.
respondent_pageviews = pd.merge(respondent_pageviews, survey_metadata["pages"][["page_id",
"page_seq"]], on="page_id", how="left")
respondent_pageviews.sort_values("page_seq", inplace=True)

# Results table.
pageviews_results = respondent_pageviews.pivot(index="respondent_id", columns="page_seq",
values=["n_pageviews", "dur_vis1",
"dur_vis1_foc",
"dur_tot_foc", "dur_tot",
"n_p_visits"])
pageviews_results.columns = [tup[0] + "_p" + str(tup[1]) for tup in
pageviews_results.columns.values]
pageviews_results.reset_index(inplace=True)
pageviews_results = pd.merge(respondent_pageviews_sum, pageviews_results, on="respondent_id")
pageviews_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]], pageviews_results,
on="respondent_id")

# Export results to Excel.
if settings["export_to_excel"]:
    pageviews_results.to_excel(settings["results_folder"] + "pageviews_respondent_level.xlsx",
index=False)

# %% RESPONSES
responses_analysis = responses_proc.copy()

# %% Response times for individual items

# Append tables of pageviews and responses to calculate response times.
# TODO: Popravi za primere, ko ne obstaja text_leave vnos, ki ga zdaj uporabljaÅ; kot Åas
odgovora na odprto vpraÅ;anje
#check.sort_values(["respondent_id", "pageview_id", "response_ts"], inplace=True)
#check["flag_repeated"] = check["response_action"].eq(check["response_action"].shift(1))
#check.loc[check["item_name"] != check["item_name"].shift(1), "flag_repeated"] = False

response_times_analysis = pd.concat([pageviews_proc[["pageview_id", "respondent_id",
"load_ts"]],
responses_proc[["response_id", "respondent_id", "pageview_id",
"response_ts", "item_name", "response_action"]]])

# We drop text field entry events here.
# TODO: Calculate time between entering and leaving the text field before dropping.
response_times_analysis = response_times_analysis[(response_times_analysis["response_action"])
!= "text_enter"]

response_times_analysis["event_type"] = np.where(response_times_analysis["load_ts"].notna(),
"pageview", np.nan)
response_times_analysis["event_ts"] = np.where(response_times_analysis["load_ts"].notna(),
response_times_analysis["load_ts"], pd.NaT)

response_times_analysis["event_type"] =
np.where(response_times_analysis["response_ts"].notna(), "response",
response_times_analysis["event_type"])
response_times_analysis["event_ts"] = np.where(response_times_analysis["response_ts"].notna(),
response_times_analysis["response_ts"], response_times_analysis["event_ts"])

response_times_analysis["event_ts"] =
response_times_analysis["event_ts"].astype(np.datetime64)

response_times_analysis = response_times_analysis[["response_id", "pageview_id",
"respondent_id", "event_type", "event_ts", "item_name", "response_action"]]
response_times_analysis.sort_values(["respondent_id", "event_ts"], inplace=True)

# Calculate response times as the difference between two consecutive events.
response_times_analysis.reset_index(inplace=True)
response_times_analysis["response_time"] =
response_times_analysis.sort_values(["respondent_id", "event_ts"]).groupby(["respondent_id"])["
event_ts"].diff().dt.total_seconds()

```

```

response_times_analysis = response_times_analysis[response_times_analysis["event_type"] !=
"pageview"]

# Merge data to the responses dataframe and list items with missing response times.
responses_analysis = pd.merge(responses_analysis, response_times_analysis[["response_id",
"response_time"]], how="left", on="response_id",
                             indicator="response_times_merge")
missing_response_times = responses_analysis[(responses_analysis["response_times_merge"] ==
"left_only") &
                                             (responses_analysis["response_action"] !=
"text_enter")]

if not missing_response_times.empty:
    processing_log.append(["responses_analysis", "medium", "missing data",
                          "Some response entries do not have valid calculated response
time.", missing_response_times])

del missing_response_times

# Results
# TODO: Check that item ID and item name are truly unique combinations and rethink if both are
needed and are OK for grouping.
# TODO: Kako je rezultat suma, ali je ena izmed setetih vrednosti missing?
respondent_item_responses = responses_analysis.groupby(["respondent_id", "item_name",
"item_id"]).agg(n_responses=("item_id", "count"),

response_time_first=("response_time", "first"),

response_time_total=("response_time", "sum")).reset_index()

respondent_item_responses["response_time_total"] =
np.where(respondent_item_responses["response_time_total"] == 0,
                                                np.nan,
respondent_item_responses["response_time_total"])
respondent_item_responses["response_time_first"] =
np.where(respondent_item_responses["response_time_first"] == 0,
                                                np.nan,
respondent_item_responses["response_time_first"])

# Number of response changes derived simply by subtracting 1 from the number of responses to
each item. (Each subsequent response to an
# item counts as a change.)
respondent_item_responses["n_response_changes"] = respondent_item_responses["n_responses"] - 1
respondent_item_responses["changed_response"] =
np.where(respondent_item_responses["n_response_changes"] > 0, True, False)

# Aggregated results at the respondent level.
respondent_responses_sum =
respondent_item_responses.groupby("respondent_id").agg(n_responses=("n_responses", "sum"),

n_response_changes=("n_response_changes", "sum"),

n_response_change_items=("changed_response", "sum")).reset_index()
# Replace item names with common names if specified.
# if settings["use_common_names"]:
#     respondent_item_responses["original_name"] = respondent_item_responses["item_name"]
#     for index, i_item in ref_common_item_names.iterrows():
#         respondent_item_responses["item_name"] =
np.where(respondent_item_responses["item_name"] == i_item["item_name"],
                                                i_item["common_name"],
#
#
respondent_item_responses["item_name"])
# del i_item
# del index

# Prepare results
responses_results = respondent_item_responses.pivot(index="respondent_id",
columns="item_name", values=["n_responses", "response_time_first", "response_time_total"])
responses_results.columns = ['_'.join(col).strip() for col in
responses_results.columns.values]
responses_results.reset_index(inplace=True)
responses_results = pd.merge(respondent_responses_sum, responses_results, on="respondent_id")
responses_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]], responses_results,
on="respondent_id")

# Export results to Excel.

```

```

if settings["export_to_excel"]:
    responses_results.to_excel(settings["results_folder"] + "responses_respondent_level.xlsx",
index=False)

# %% POINTER MOVEMENTS
pointer_analysis = pointer_proc.copy()

# Move duration and speed.
# TODO: Not sure if move speed really makes sense, eespecially with aggregations at the level
of page or respondent.
pointer_analysis["move_duration"] = (pointer_analysis["move_end_ts"] -
pointer_analysis["move_start_ts"]).dt.total_seconds()
pointer_analysis["move_speed"] = pointer_analysis["move_distance"] /
pointer_analysis["move_duration"]

### Respondent-level
# Aggregation at the page level.
respondent_pointer_page = pointer_analysis.groupby(["respondent_id",
"page_id"]).agg(move_duration=("move_duration", "sum"),

move_distance=("move_distance", "sum")).reset_index()
respondent_pointer_page["move_speed"] = respondent_pointer_page["move_distance"] /
respondent_pointer_page["move_duration"]

# Aggregation at the respondent level.
respondent_pointer_sum =
pointer_analysis.groupby("respondent_id").agg(move_dur_tot=("move_duration", "sum"),

move_distance_total=("move_distance", "sum")).reset_index()
respondent_pointer_sum["move_speed_total"] = respondent_pointer_sum["move_distance_total"] /
respondent_pointer_sum["move_dur_tot"]

# Add page order.
respondent_pointer_page = pd.merge(respondent_pointer_page,
survey_metadata["pages"][["page_id", "page_seq"]], on="page_id", how="left")
respondent_pointer_page.sort_values("page_seq", inplace=True)

# Results table.
pointer_results = respondent_pointer_page.pivot(index="respondent_id", columns="page_seq",
values=["move_duration", "move_distance", "move_speed"])
pointer_results.columns = [tup[0] + "_p" + str(tup[1]) for tup in
pointer_results.columns.values]
pointer_results.reset_index(inplace=True)
pointer_results = pd.merge(respondent_pointer_sum, pointer_results, on="respondent_id")
pointer_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]], pointer_results,
on="respondent_id")

# Export to Excel
if settings["export_to_excel"]:
    pointer_results.to_excel(settings["results_folder"] +
"pointer_movement_respondent_level.xlsx", index=False)

# %% CLICKS
clicks_analysis = clicks_proc.copy()

### Respondent-level
# Aggregation at the page level.
respondent_clicks_page = clicks_analysis.groupby(["respondent_id",
"page_id"]).agg(n_clicks=("click_id", "count")).reset_index()

# Aggregation at the respondent level.
respondent_clicks_sum = clicks_analysis.groupby("respondent_id").agg(n_clicks=("click_id",
"count")).reset_index()

# Add page order.
respondent_clicks_page = pd.merge(respondent_clicks_page, survey_metadata["pages"][["page_id",
"page_seq"]], on="page_id", how="left")
respondent_clicks_page.sort_values("page_seq", inplace=True)

# Results table.
clicks_results = respondent_clicks_page.pivot(index="respondent_id", columns="page_seq",
values=["n_clicks"])
clicks_results.columns = [tup[0] + "_p" + str(tup[1]) for tup in
clicks_results.columns.values]
clicks_results.reset_index(inplace=True)

```

```

clicks_results = pd.merge(respondent_clicks_sum, clicks_results, on="respondent_id")
clicks_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]], clicks_results,
on="respondent_id")

# Export to Excel
if settings["export_to_excel"]:
    clicks_results.to_excel(settings["results_folder"] + "clicks_respondent_level.xlsx",
index=False)

# %% MESSAGES
messages_analysis = messages_proc.copy()

### Respondent-level

# Aggregation at the respondent level.
respondent_messages_sum = messages_analysis.groupby("respondent_id").agg(n_messages=("msg_id",
"count")).reset_index()
respondent_inr_messages_sum = messages_analysis[messages_analysis["msg_type"] ==
"item_nonresponse"].groupby("respondent_id").agg(n_inr_messages=("msg_id",
"count")).reset_index()
respondent_messages_sum = pd.merge(respondent_messages_sum, respondent_inr_messages_sum,
how="left", on="respondent_id")
del respondent_inr_messages_sum

# Results
messages_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]],
respondent_messages_sum, on="respondent_id")

# Export to Excel
if settings["export_to_excel"]:
    messages_results.to_excel(settings["results_folder"] + "messages_respondent_level.xlsx",
index=False)

# %% VIEW CHANGES
# TODO: This only analyses orientation changes, it needs to be completely rewritten to account
for other view change events.
view_changes_analysis = view_changes_proc.copy()
respondent_view_changes_sum =
view_changes_analysis.groupby("respondent_id").agg(n_orient_change_pages=("page_id",
"nunique")).reset_index()
view_changes_results = pd.merge(ref_recnum_id[["respondent_id", "recnum"]],
respondent_view_changes_sum, on="respondent_id")

if settings["export_to_excel"]:
    view_changes_results.to_excel(settings["results_folder"] +
"view_changes_respondent_level.xlsx", index=False)

# %% SUMMARY EXPORTS

# Aggregations at the respondent level.
respondent_level_sum = pd.merge(ref_recnum_id[["respondent_id", "recnum"]],
respondent_pageviews_sum, on="respondent_id")
respondent_level_sum = pd.merge(respondent_level_sum, respondent_responses_sum,
on="respondent_id", how="outer")
respondent_level_sum = pd.merge(respondent_level_sum, respondent_messages_sum,
on="respondent_id", how="outer")
respondent_level_sum = pd.merge(respondent_level_sum, respondent_clicks_sum,
on="respondent_id", how="outer")
respondent_level_sum = pd.merge(respondent_level_sum, respondent_pointer_sum,
on="respondent_id", how="outer")
respondent_level_sum = pd.merge(respondent_level_sum, respondent_view_changes_sum,
on="respondent_id", how="outer")
respondent_level_sum = pd.merge(respondent_level_sum, respondent_focus_out_sum,
on="respondent_id", how="outer")
respondent_level_sum.to_excel(settings["results_folder"] + "respondent_level_sum.xlsx",
index=False)
ref_item_names.to_excel(settings["results_folder"] + "ref_item_names.xlsx", index=False)

del respondent_pageviews
del respondent_visits
del respondent_pageviews_sum

```