Zespołowy projekt studencki

Klasyfikacja sygnałów SSVEP przy wykorzystaniu metod uczenia maszynowego

Katarzyna Filipiuk
Krzysztof Piwoński
Urszula Romaniuk
Izabela Szopa

Wstęp

Praca ta stworzona została w semestrze letnim roku akademickiego 2017/2018 na Wydziale Fizyki Uniwersytetu Warszawskiego. Jej autorami są Katarzyna Filipiuk, Krzysztof Piwoński, Urszula Romaniuk oraz Izabela Szopa. Nadzór merytoryczny sprawował doktor habilitowany Jarosław Żygierewicz.

Celem niniejszego projektu jest sprawdzenie jakości klasyfikacji danych EEG, pochodzacych z interfejsu mózg-komputer (ang. *brain-computer interface*, BCI). W analizowanym eksperymencie koncepcją działania BCI są potencjały wywołane stanu ustalonego (ang. *Steady State Visual Evoked Potentials*, SSVEP). Przeprowadzona analiza ukazuje możliwości wykorzystania lasów losowych oraz lasów kernelowych w analize danych SSVEP. Istotną częścią przeprowadzonej analizy są również wykorzystywane w lasach losowych cechy oraz funkcja odległości do porównywania sygnałów. Ponadto ukazane zostało wykorzystanie lasów kernelowych w kontekście potencjalnych zysków z korzystania z tego modelu.

Dane

Dane analizowane w tym projekcie zostały udostępnione przez Annę Chabudę. Zostały one zebrane na potrzeby pracy tejże autorki pod tytułem "Interfejs mózg-komputer wykorzystujący informację o fazie bodźca SSVEP - implementacja i badanie wydajności". Przebieg badania oraz opis interfejsu jest przedstawiony w przytoczonej pracy.

W zbiorze danych znajdują się sygnały z sesji kalibracyjnej pochodzące od 14 osób. Każda z nich stymulowana była dziesięcioma różnymi częstościami pochodzącymi ze zbioru {30, 31, 32, 33, 34, 35, 36, 37, 38, 39}.

Sesja kalibracyjna polegała na skupianiu uwagi osoby badanej na wybranym polu ekranu oznaczonym gwiazdką. Ekran składał się z ośmiu niezależnych od siebie pól. Każde z nich podświetlane było z inną częstością. Eksperyment składał się z 80 stymulacji, z których każda trwała 5 sekund. Po każdym wyświeleniu następowała 1 sekunda przerwy. Częstość stymulacji była dobierana losowo. W całym badaniu każda z 10 wybranych częstości była powtórzona 8 razy.

Sygnał EEG pochodził z ośmiu elektrod wodnych z systemu 10-10 {Cz, Pz, P03, P04, P07, P08, O1, O2}. Elektroda uziemiająca znajdowała się na lewym obojczyku osoby badanej. Dodatkowo zbierany był sygnał z diod podświetlających ekran. Pomiar był próbkowany z częstością 512 Hz.

Do ekstrakcji danych wykorzystane zostały programy udostępnione na licencji GPL, pochodzące ze strony https://github.com/BrainTech/openbci/tree/mgr_ssvep/analysis/mgr_ssvep (https://github.com/BrainTech/openbci/tree/mgr_ssvep/analysis/mgr_ssvep).

Ekstrakcja danych

Program został napisany w środowisku *Python 3*. Korzysta on z pakietu *Numpy* udostępnionego na licencji BSD oraz z pakietów Anny Chabudy udostępnionych na licencji GPL.

In [1]:
```
# -*- coding: utf-8 -*-
# !/usr/bin/env python

import numpy as np

from obci_signal_processing import read_manager
import filters as SPF
import parse_signal as SPPS
import montage_signal as SPSM
```

Zdefiniowano kanały niezawierające sygnałów EEG (CHANNELS_TO_IGNORE). Do montażu użyto kanału "P07" co zapisano pod zmienną CHANNELS_TO_MONTAGE. Ze względu na strukturę klas znajdujących się w zapożyczonych pakietach, niezbędne jest definiowanie zmiennej CHANNELS_TO_LEAVE, która jednakże jest nieistotna z punktu widzenia niniejszego programu. Zmienna MONTAGE_TYPE opisuje typ montażu. W tym przypadku zastosowany został montaż jednobiegunowy. FREQ_TO_TRAIN zawiera listę częstości, którymi stymulowane były osoby badane. Koniecznym jest przekazanie listy identyfikatorów osób badanych, ponieważ na tej podstawie następuje ekstrakcja sygnału.

In [2]:
```
CHANNELS_TO_IGNORE = [u'Dioda1', u'Dioda2', u'Dioda3', u'Dioda4',
                      u'Dioda5', u'Dioda6', u'Dioda7', u'Dioda8', u'Am
pSaw', u'DriverSaw']

CHANNELS_TO_MONTAGE = [u'P07']

CHANNELS_TO_LEAVE = []

MONTAGE_TYPE = 'diff'

FREQ_TO_TRAIN = [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]

PEOPLE = [u'ania2', u'ania_maria', u'anna_piatek', u'iza', u'karol2',
u'karol_wielki', u'maciek', u'magdalena',
          u'mateusz_b', u'mati', u'max', u'PIOTR', u'piotrrrr', u'sas
z', u'tomasz']
```

Zadaniem poniższej klasy jest wyodrębnienie sygnałów dla wybranej osoby z sesji kalibracyjnej. W tym celu do instancji przekazywana jest nazwa użytkownika. Jako parametr domyślny został zdefiniowany katalog zawierający surowe dane, liczbę powtórzeń każdej częstości oraz słowa kluczowe używane w pliku z rozszerzeniem ".tag". Sprecyzowane zostały także parametry sygnału EEG, takie jak nazwy kanałów bez sygnału, typ montażu, elektroda referencyjna oraz częstości występujące w sesji kalibracyjnej.

Wszystkie metody poza signal_extraction są pomocnicze i nie jest potrzebne wywoływanie ich na poziomie instancji. Metoda signal_extraction zwraca słownik dla wybranej osoby, którego kluczami są częstości stymulacji. Wartością pojedynczego klucza jest trójwymiarowa macierz, która zawiera sygnał EEG z podziałem na kanały i powtórzenia.

In [3]:
```
class ExtractSignal(object):
    def __init__(self,
                 person,
                 file_1=u'ssvep_calibration_data/ssvep_pattern_',
                 file_2=u'_calibration',
                 l_trial=5,
                 ignore_channels=CHANNELS_TO_IGNORE,
                 montage_type=MONTAGE_TYPE,
                 montage_channels=CHANNELS_TO_MONTAGE,
                 leave_channels=CHANNELS_TO_LEAVE,
                 tag_name=u'diodes',
                 freq_to_train=FREQ_TO_TRAIN):

        super(ExtractSignal, self).__init__()
        self.file_1 = file_1
        self.file_2 = file_2
        self.person = person
        self.mgr = self._init_read_manager()
        self.channels_gains = self.mgr.get_param('channels_gains')
        self.all_channels = self.mgr.get_param('channels_names')
        self.fs = float(self.mgr.get_param("sampling_frequency"))
        self.l_trial = l_trial
        self.ignore_channels = ignore_channels
        self.montage_type = montage_type
        self.montage_channels = montage_channels
        self.tag_name = tag_name
        self.freq_to_train = freq_to_train
        self.use_channels = self._init_channels_names()
        self.leave_channels = leave_channels
        self.montage_matrix = self._init_montage_matrix(self.all_chann
els,
                                                        self.use_chann
els,
                                                        self.montage_t
ype,
                                                        self.montage_c
hannels,
                                                        self.leave_cha
nnels)

    def _init_read_manager(self):  #
        file_name = self.file_1 + self.person + self.file_2
        return read_manager.ReadManager(file_name + '.obci.xml',
                                        file_name + '.obci.raw',
                                        file_name + '.obci.tag')


    def _init_channels_names(self):
        use_channels = []

        for ch_name in self.mgr.get_param('channels_names'):
            if ch_name not in self.ignore_channels and \
                            ch_name not in self.montage_channels:
                use_channels.append(ch_name)
        return use_channels

    def _init_montage_matrix(self, all_channels, use_channels, montage
```

```
_type,
                                montage_channels, leave_channels):
        return SPSM.get_montage_matrix(all_channels, use_channels, mon
tage_type,
                                       montage_channels, leave_channel
s)

    def _signal_segmentation(self, mgr, l_trial, offset, tag_name):
        return SPPS.signal_segmentation(mgr, l_trial, offset, tag_name
)

    def _highpass_filtering(self, signal, use_channels, all_channels,
fs):  #
        return SPF.highpass_filter(signal, use_channels, all_channels,
 fs)

    def _bandpass_filtering(self, signal, use_channels, all_channels,
fs):  #
        return SPF.cheby2_bandpass_filter(signal, use_channels, all_ch
annels, fs)

    def _montage_signal(self, signal, montage_matrix):  #
        return SPSM.montage(signal, montage_matrix)

    def _signal_processing(self, signal):  # self.channels_gains, sel
f.montage_matrix  self.fs

        # 0. to volts
        signal = self._to_volts(signal, self.channels_gains)

         # 1. cutof mean signal
        signal = self._highpass_filtering(signal,
                                          sum([self.use_channels,
                                               self.montage_channels],
 []),
                                          self.all_channels,
                                          self.fs)

        # 2. bandpass filtering (ss.cheby2(3, 50,[49/(fs/2),50/(fs/
2)], btype='bandstop',
        #                       analog=0))

        signal = self._bandpass_filtering(signal,
                                          sum([self.use_channels,
                                               self.montage_channels],
 []),
                                          self.all_channels,
                                          self.fs)

        # 3. montage signal
        signal = self._montage_signal(signal,
                                      self.montage_matrix)

        return signal

    def _to_volts(self, signal, channels_gains):  #
        return SPPS.to_volts(signal, channels_gains)
```

```
    def signal_extraction(self):
        # 1) create smart tag object
        smart_tags = self._signal_segmentation(self.mgr, self.l_trial,
 0, self.tag_name)
        # 2)

        # zwracana macierz ma wymiary: len(use_channels) + len(leave_c
hannels) (15), obciete probki po montazu (2311),
        # powtorzenia eksperymentu w kolejnosci (wszytskie 80)?

        temp = self._signal_processing(smart_tags[0].get_samples())

        counter_dict = {f:0 for f in self.freq_to_train}
        signal_dict = {f:np.zeros((temp.shape[0], temp.shape[1], int(l
en(smart_tags) / len(self.freq_to_train))))
                       for f in self.freq_to_train}
        frequency_list = np.zeros(len(smart_tags))
        for ind in range(len(smart_tags)):
            f = int(smart_tags[ind].get_tags()[0]['desc']['freq'])
            signal_dict[f][:, :, counter_dict[f]] = self._signal_proce
ssing(smart_tags[ind].get_samples())
            frequency_list[ind] = smart_tags[ind].get_tags()[0]['desc'
]['freq']
        return signal_dict
```

Poniższy kod przedstawia przykład zastosowania wyżej opisanej klasy.

In [4]:
```
# ania2.signal_extraction() - slownik o kluczach takich jak 'FREQ_TO_TRAIN' czyli czestosciach pobudzania
# pod odpowiednim kluczem znajduje się macierz o wymiarach [kanaly : probki : powtorzenia danej czestosci dla tej osoby]
# kolejnosc kanalow to ['O1', 'O2', 'PO8', 'PO3', 'PO4', 'Pz', 'Cz',
 'Dioda1', 'Dioda2', 'Dioda3', 'Dioda4', 'Dioda5',
# 'Dioda6', 'Dioda7', 'Dioda8'] gdzie nazwy elektrod sa zapisane w self.use_channels a nazwy diod w self.leave_channels
# w kalibracji zawsze z zadana czestoscia mrugala Dioda1

ania2 = ExtractSignal(PEOPLE[0])
eksperyment_ania2 = ania2.signal_extraction()

# pierwsze powtorzenie eksperymentu dla czestosci 34 Hz (wszytskie kanaly)
eksperyment_ania2[34][:, :, 0]

# pierwsze powtorzenie eksperymentu dla czestosci 34 Hz (kanal O2)
eksperyment_ania2[34][1, :, 0]

print(ania2.use_channels)
```

```
2018-09-18 11:26:51,197 - smart_tags_manager - INFO - Start initialising smart tags.
2018-09-18 11:26:51,201 - smart_tags_manager - INFO - Finished initialising smart tags.
```

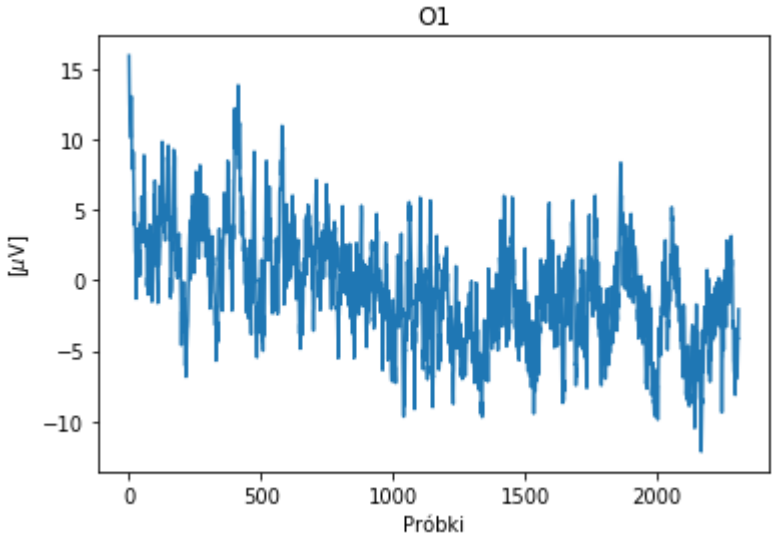
```
[u'O1', u'O2', u'PO8', u'PO3', u'PO4', u'Pz', u'Cz']
```

Ta część kodu przedstawia wyświetlenie surowego sygnału EEG ze wszystkich kanałów po wykonaniu referencji. Sygnał pochodził z pierwszego powtórzenia stymulacją o częstości 34 Hz.

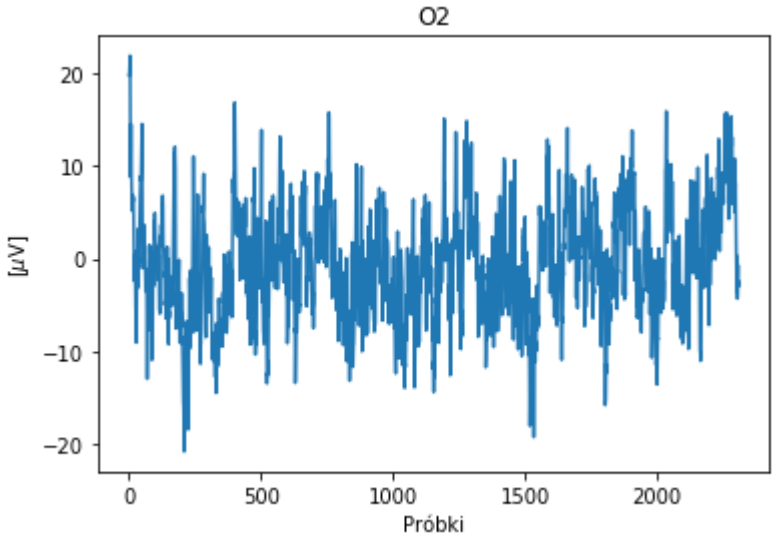
In [29]:
```
import matplotlib.pyplot as plt

channels = list(ania2.use_channels)
print(channels)

for i in range(len(channels)):
    plt.title(channels[i])
    plt.plot(eksperyment_ania2[34][i, :, 0])
    plt.xlabel(u"Próbki")
    plt.ylabel("[$\mu$V]")
    plt.show()
```

```
[u'O1', u'O2', u'PO8', u'PO3', u'PO4', u'Pz', u'Cz']
```

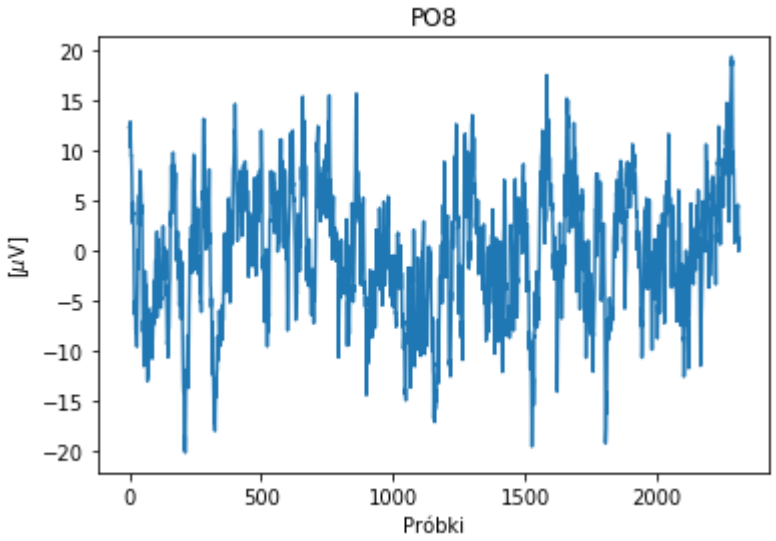
Out[29]:



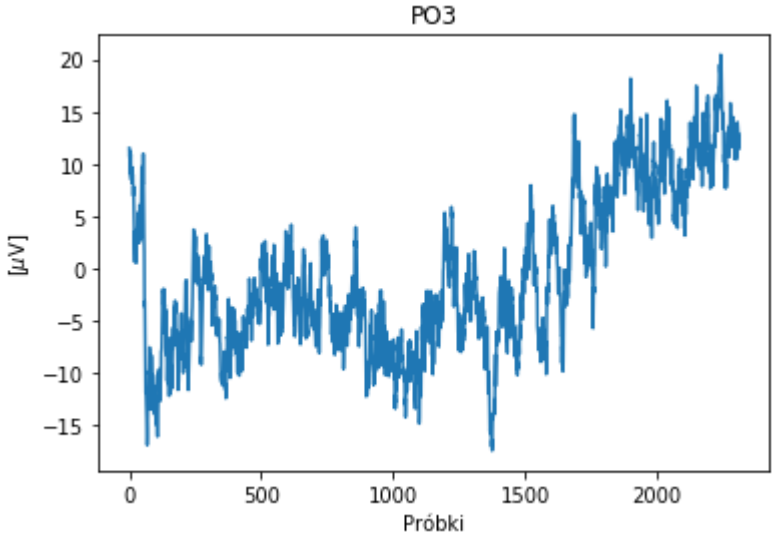
Out[29]:



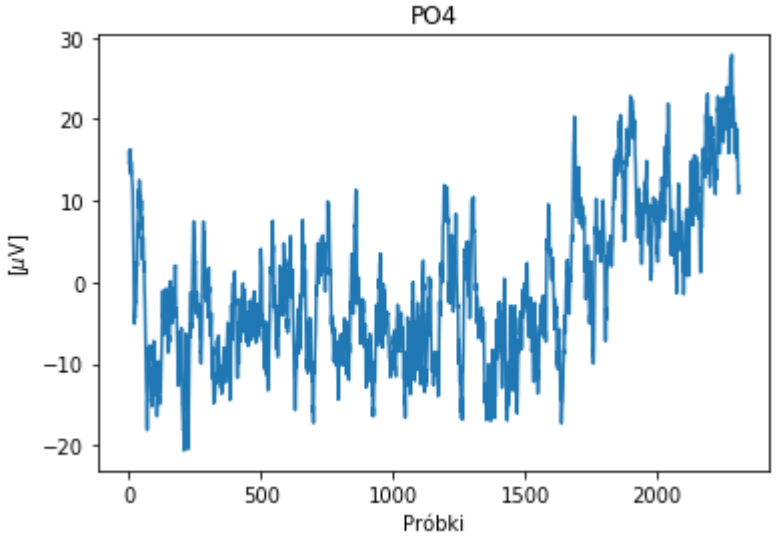
Out[29]:



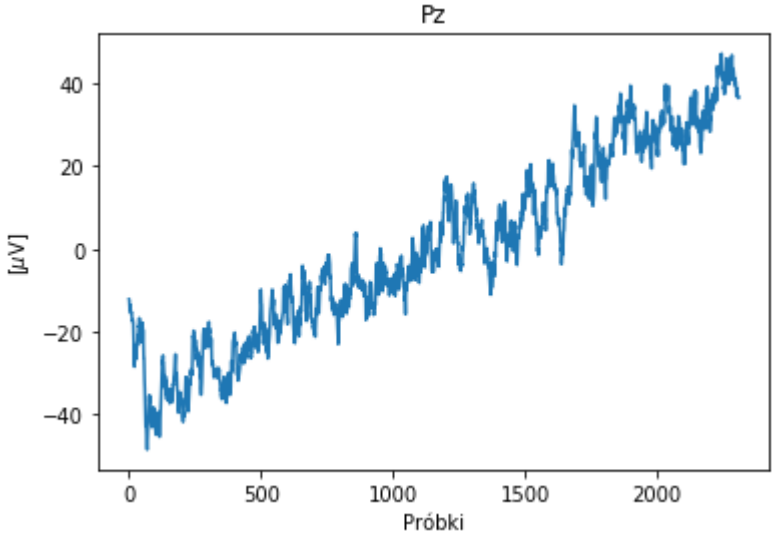
Out[29]:



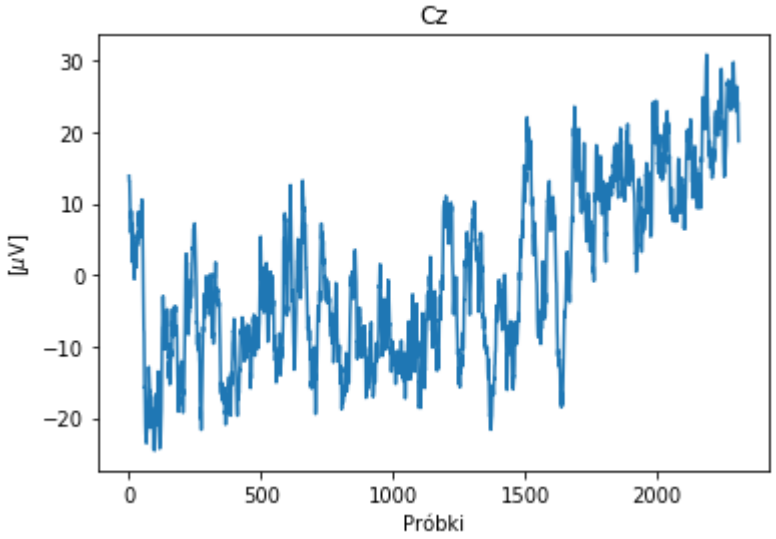
Out[29]:



Out[29]:

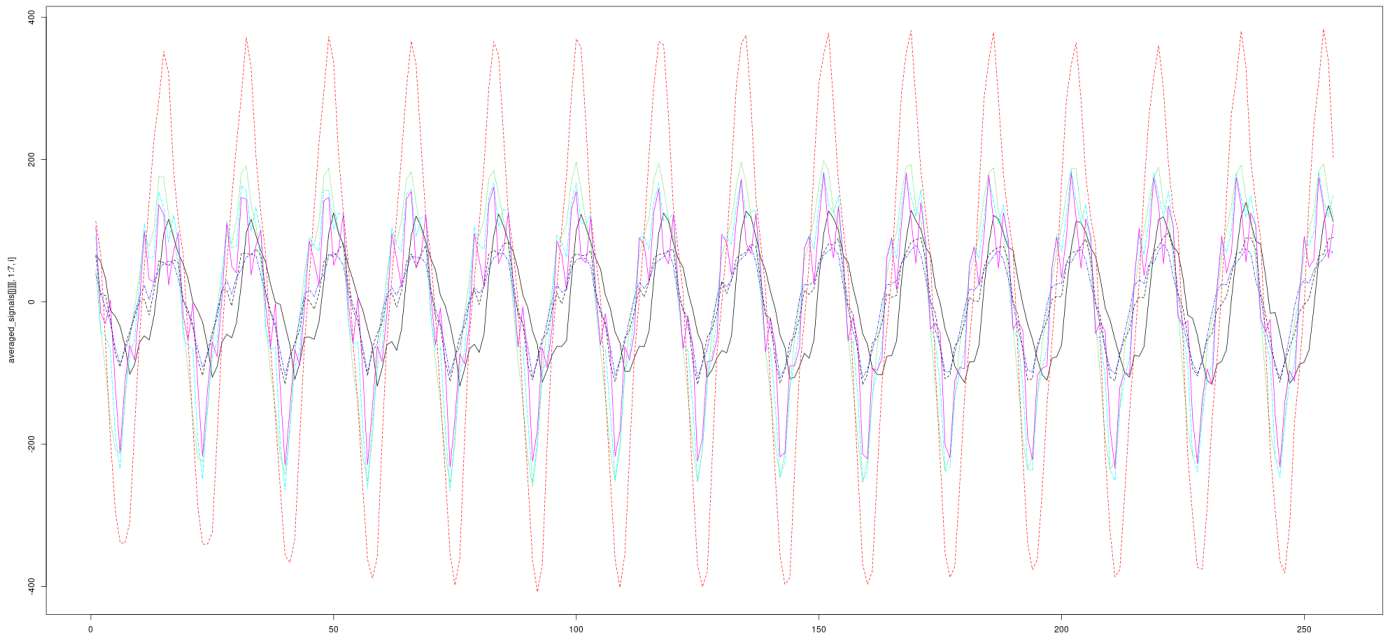


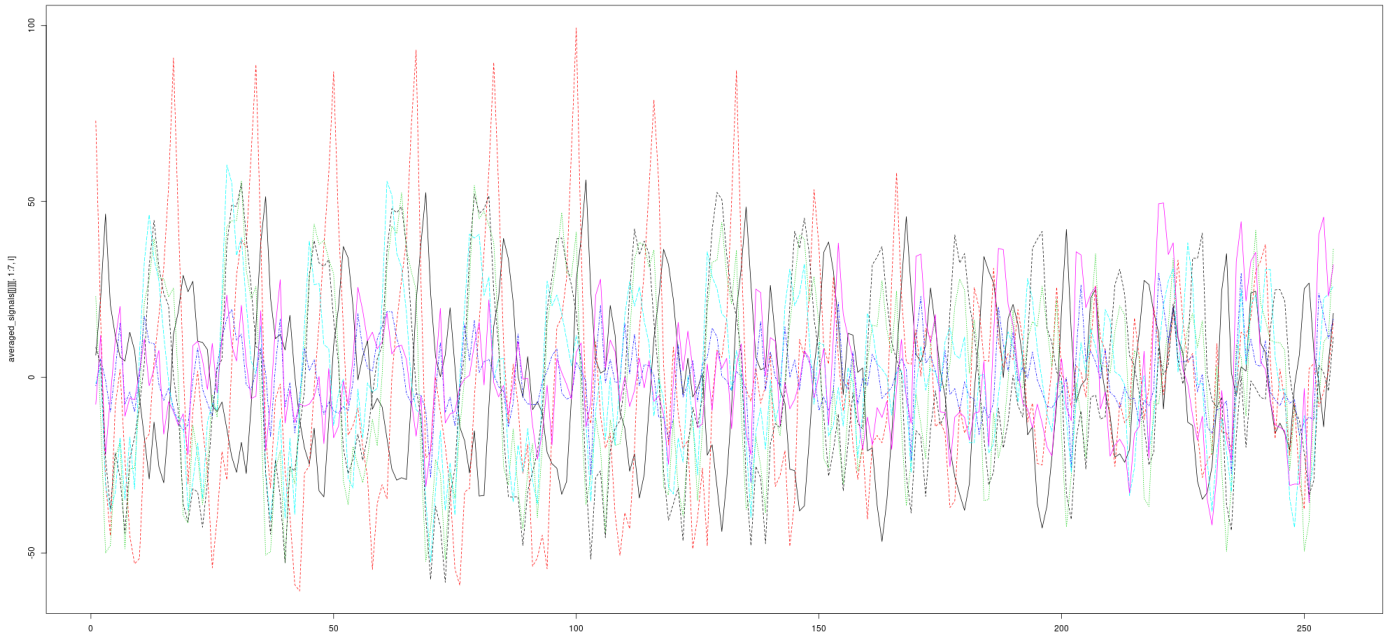
Out[29]:



Uśrednianie sygnału w fazie

Po wykonaniu ekstrakcji danych sygnały zostały uśrednione w fazie w podobny sposób jak w pracy *High Frequency SSVEP-BCI With Hardware Stimuli Control and Phase-Synchronized Comb Filter* A. Chabuda et. al. W porównaniu do wyżej wymienionej pracy zamiast usuwać liniowe komponenty sygnału odjęto średnią kroczącą w każdym punkcie od wartości sygnału w tym punkcie. Poniżej został zamieszczny przykład sygnału po uśrednieniu w czasie dla częstości uśredniania równej częstości decyzji i w przypadku gdy częstość uśredniania jest różna od częstości decyzji oraz kod wykorzystany do wykonania uśredniania.





In [0]:
```
library(h5)
library(parallel)
ma <- function(x,n=5){filter(x,rep(1/n,n), sides=2)}

f <- h5file("ssvep.hdf5", "r")
Q <- f["data"][]


averaged_signals <- mclapply(30:39, function(freq) {
  w <- apply(Q, c(1,3), function(signal) {
    signal <- signal-ma(signal, 31)
    signal <- signal[!is.na(signal)]

    rowSums(
      sapply(0:100, function(i) {
        signal[(round(512/freq*i)+1):(round(512/freq*i)+256)]
      })
    )
  })
}, mc.cores=10)
```

Analiza danych

W przeprowadzonej analizie wykonano klasyfikację przy wykorzystaniu modelu random forest oraz lasów kernelowych. Dla każdego uczestnika badania oraz dla każdej częstości tworzono oddzielny klasyfikator binarny. Przykładowo dla 1. uczestnika tworzono model mający określić czy uczestnik koncentruje się na polu migającym z częstością 30 Hz. Wynikiem każdego klasyfikatora było prawdopodobieństwo takiego zdażenia. Jako miarę jakości każdego klasyfikatora binarnego wybrano AUROC (Area under ROC curve). Na tej podstawie stworzono mapy reprezentujące pewność klasyfikacji w zależności od uczestnika badania oraz częstości.

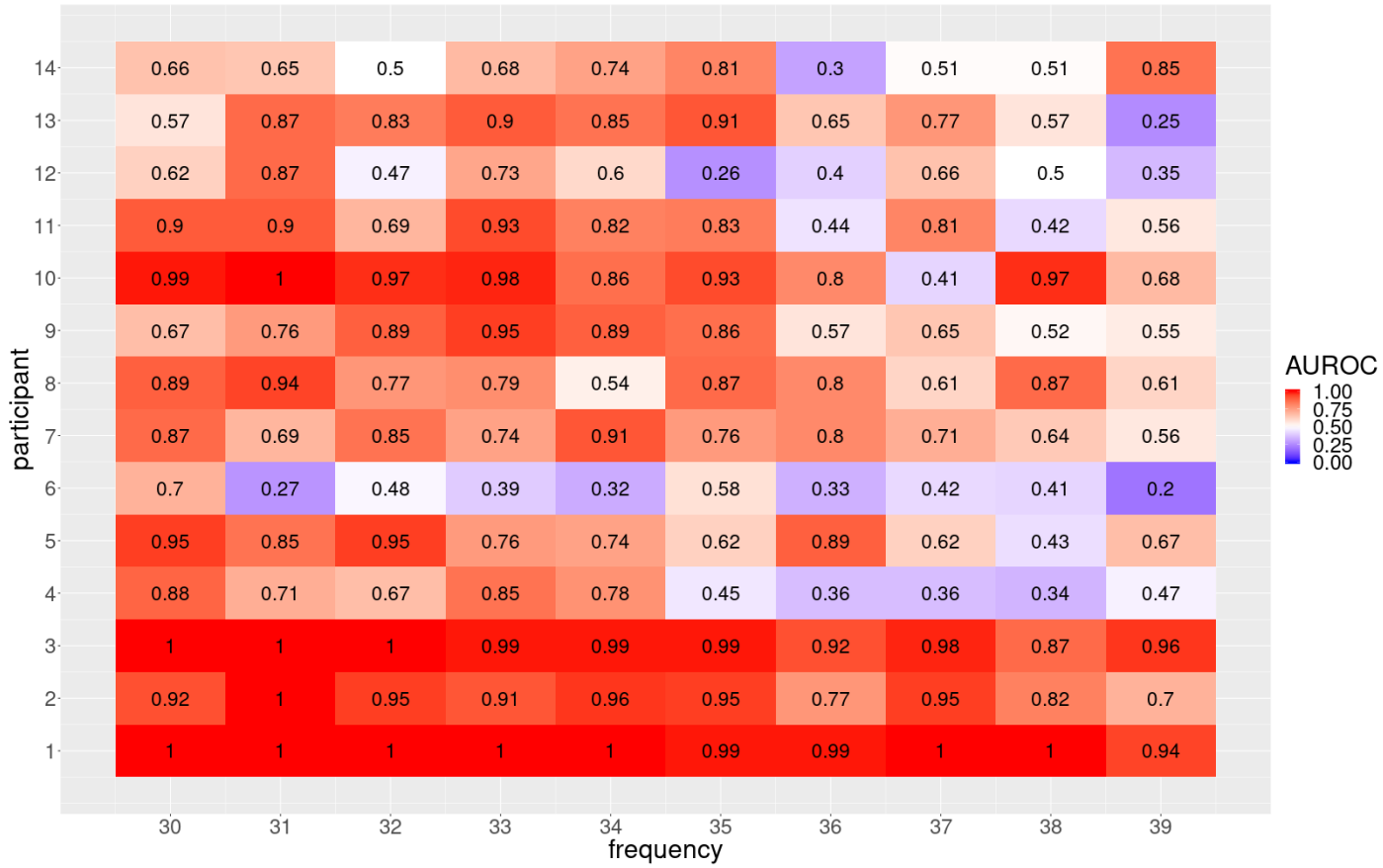
Metodologia

W poniższej analizie korzystano z lasów losowych (https://link.springer.com/article/10.1023%2FA%3A1010933404324 (https://link.springer.com/article/10.1023%2FA%3A1010933404324)). Las losowy jest modelem składającym się z wielu drzew decyzyjnych. Każde z drzew w lesie głosuje osobno podczas predykcji. Aby głosowanie działało drzewa w lesie muszą być wzajemnie możliwie nieskorelowane. W tym celu stosuje się bagging (bootstrap aggregating). Każde drzewo jest uczone na podzbiorze przypadków. Podzbiór przypadków jest tworzony poprzez losowanie ze zwracaniem przypadków z wejściowego zbioru. Zbiór treningowy dla każdego drzewa jest wielkości zbioru wejściowego. Poza redukcją wariancji wykorzystanie tylko części przypadków do konstrukcji drzewa pozwala na wykorzystanie reszty przypadków do weryfikacji tego drzewa (*out of bag error*). Przy małej liczbie przypadków taki sposób oszacowywania błędu jest kluczowy, ponieważ pozwala na oszacowanie błędu bez przeznaczania części danych na kroswalidację. Dodatkowo w celu jeszcze większego zróżnicowania populacji drzew do konstrukcji węzła drzewa wykorzystuje się tylko określoną liczbę losowo wybranych cech – najczęściej pierwiastek z M, gdzie M to liczba cech.

Modyfikacją lasów losowych są lasy kernelowe. Różnicą w działaniu pomiędzy lasami losowymi, a lasami kernelowymi jest to, że węzły w drzewach poza progiem decyzji zawierają wzorce oraz funkcję odległości. Dzięki temu cechy mogą być generowane lokalnie w węźle, co umożliwia wykorzystywanie szerszej przestrzeni cech.

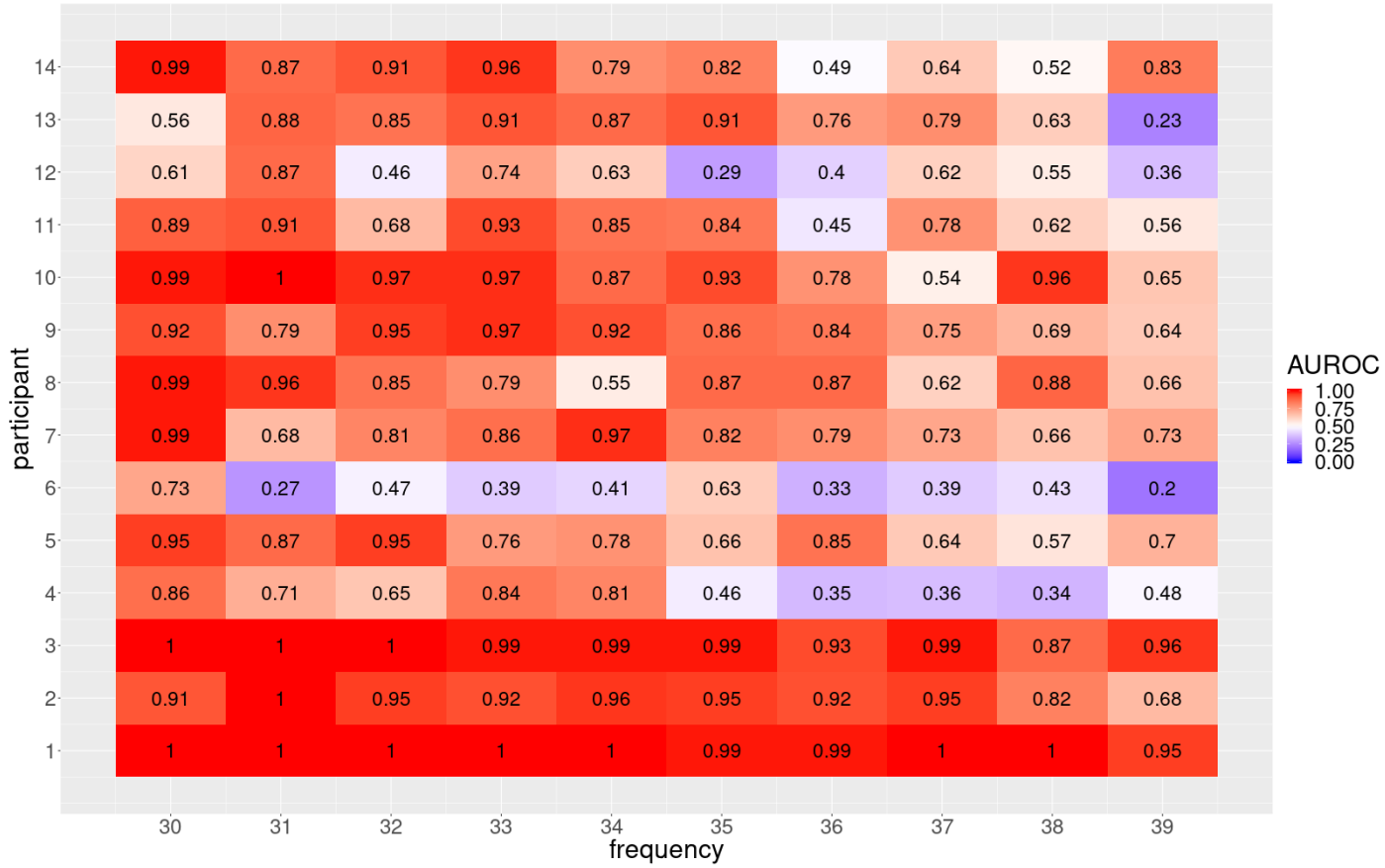
Random forest

Do generowania cech wykorzystano wszystkie kanały eeg. W dalszej częsci analizy wykorzystywano wyłącznie sygnały uśrednione w fazie. Każdy klasyfikator wykorzystywał sygnały uśrednione w czasie w badanej w danym klasyfikatorze częstości. Na przykład klasyfikator dla 1. uczestnika dla częstości 30 Hz wykorzysta dane uśrednione w fazie z częstością 30 Hz. Przypadkiem wykorzystanym w dalszej częsci analizy jest 7 kanałowy zapis uśredniony w fazie posiadający dla każdego kanału 256 próbek (odpowiada to połowie sekundy). Dla danego przypadku liczono odległość pomiędzy sygnałem z danego kanału, a sygnałem z tego samego kanału ze wszystkich przypadków dla danego uczestnika. Daje to 80 cech dla jednego kanału dla danego przypadku. Jako funkcję odległości między sygnałami wykorzystano korelację wzajemną z maksymalnym przesunięciem wynoszącym 20 probek. Biorąc pod uwagę, że wykorzystano 7 kanałów sumarycznie daje to 560 cech dla danego przypadku. Poniżej zaprezentowano rysunek, na którym został przedstawiony AUROC z predykcji *out of bag votes* klasyfikatora dla każdego uczestnika badania i każdej częstości.



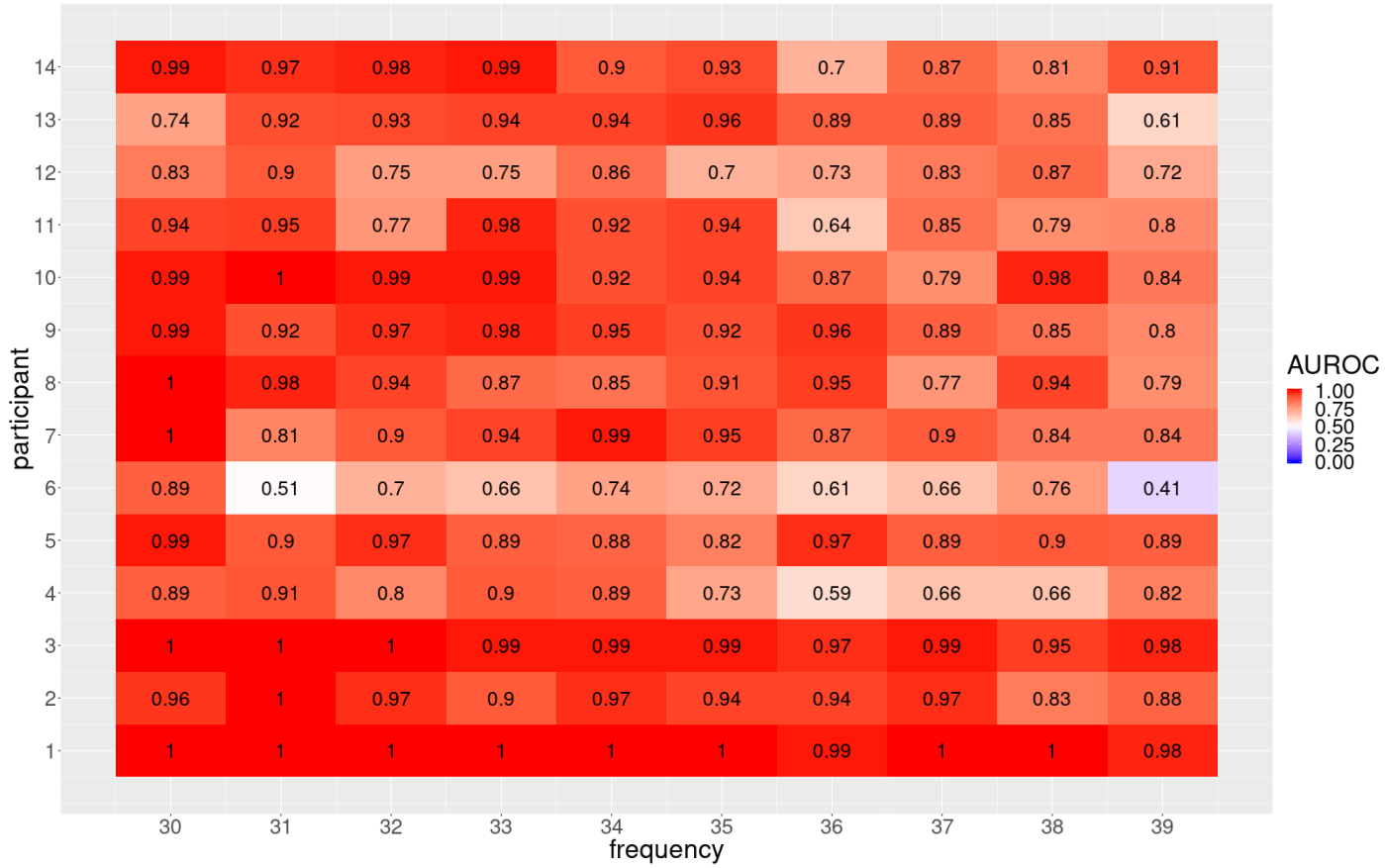
Dla uczestników nr 1, 2, 3, 10 jakość klasyfikacji jest na bardzo wysokim poziomie. W przypadku pozostałych uczestników jakość klasyfikacji jest umiarkowana lub słaba. Szczególnie niepokojące są wartości AUROC poniżej 0.5, które są gorsze od losowych klasyfikatorów. Prawdopodbnie wartości te biorą się z dominacji jednych częstości nad innymi. Uczestnicy poza częstością, na której skupiali uwagę widzieli również inne częstości na ekranie. W efekcie klasyfikator może dany zapis silniej korelować z inną częstością widoczną w zapisie niż częstość, na której koncentrował uwagę uczestnik badania.

W przeprowadzonej analizie wykonano również klasyfikację, w której poza wymienionymi wyżej cechami wykorzystano również w modelu korelację między wszystkimi kanałami zapisów, co daje dodatkowe 21 cech. Poniższy rysunek w sposób analogiczny prezentuje rezultat tej klasyfikacji.



Dodatkowe cechy mogą poprawiać jakość klasyfikacji. Na przykład dla uczestnika 14 wartości AUROC są znacząco wyższe. Prawdpopobnie korelacja między kanałami zawiera dodatkową informację, która pozwala na uzyskanie lepszego rezultatu.

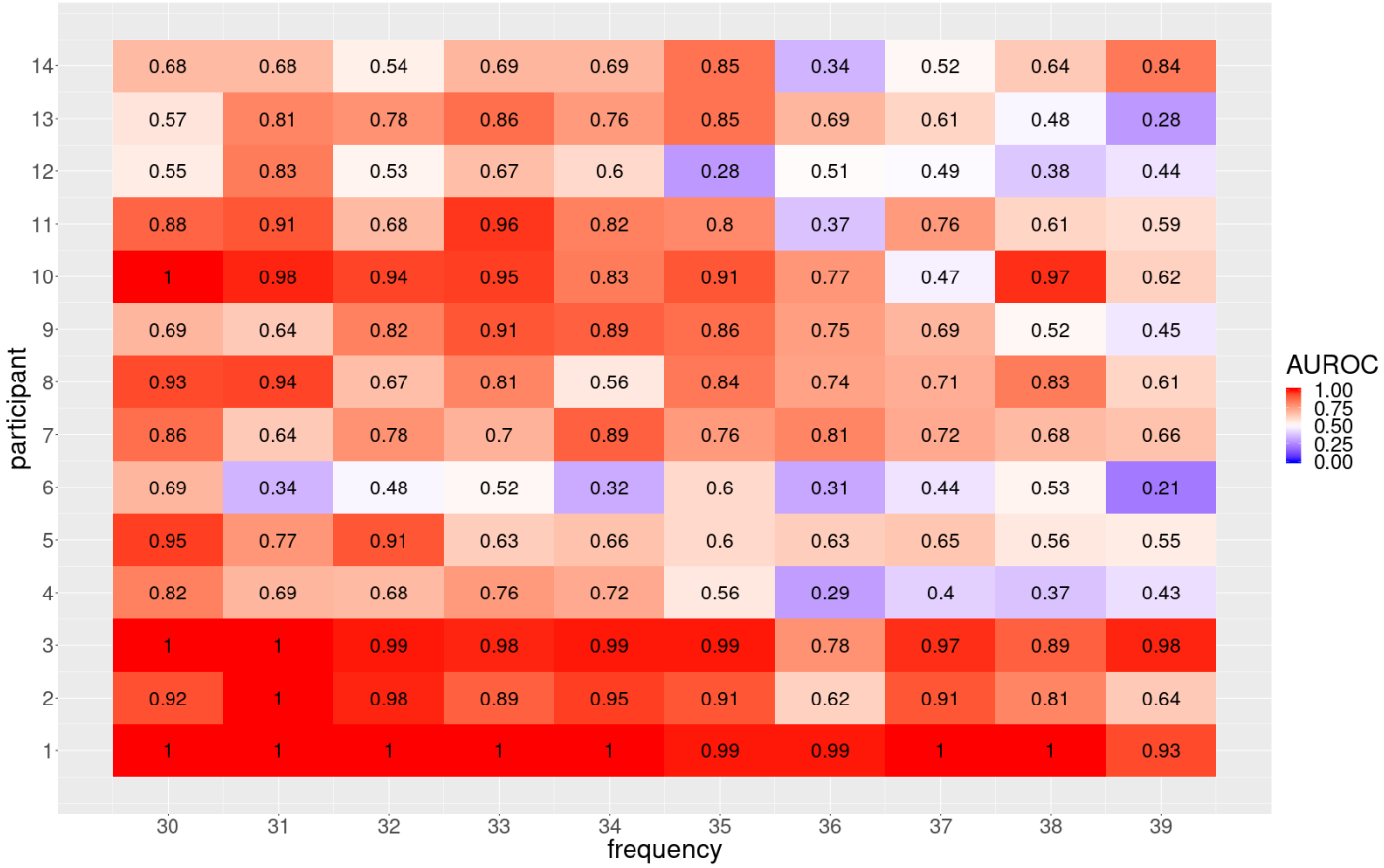
Szukając przyczyny występowania paradoksów w danych stworzono klasyfikację wykorzystującą jako wzorce jedynie przypadki, w których uczestnik miał sie koncentrować na badanej częstości. Stąd daje to 8 cech dla jednego kanału dla danego przypadku oraz sumarycznie dla 7 kanałów 56 cech dla danego przypadku. Dodatkowo wykorzystano, jak w poprzednio przeprowadzonej klasyfikacji, 21 cech związanych z korelacjami między kanałami.



Wykorzystanie mniejszej, lepiej dobranej przestrzeni cech poprawia jakość klasyfikacji. Jednocześnie pozwala na pozbycie się paradoksów występujących uprzednio w danych (wartości AUROC mniejsze niż 0.5 nie występują).

Las kernelowy

Analogicznie jak w przypadku lasu losowego przypadkiem jest 7 kanałowy sygnał eeg uśredniony odpowiednio w fazie dla badanej częstości (co daje długość 256 próbek dla każdego kanału). Las kernelowy poza progiem decyzji w węzłach dodatkowo posiada wzorzec. W przeprowadzonej analizie jako wzorce wykorzystano sygnał (po uśrednieniu w fazie) z jednego kanału. Dodatkowo własnością węzłów w drzewie jest kanał, z którego pochodzi dany wzorzec. Dzięki temu w każdym węźle drzewa dla przypadku podlegającego klasyfikacji jest wykorzystywany ten sam kanał co kanał, z którego pochodzi wzorzec. Miarą odległości wykorzystywaną w węzłach drzew jest maksymalna korelacja wzajemna z przesunięciem maksymalnie o 20 próbek między wzorcem w węźle, a sygnałem. Poniżej zaprezentowano rysunek, na którym został przedstawiony AUROC dla każdego uczestnika badania i każdej częstości. Prawdopobieństwa wykorzystane do mierzenia AUROC są liczone przez model na podstawie *out of bag votes*.



Wynik klasyfikacji jest bardzo podobny do wyniku lasu losowego mających jako cechy wyłącznie korelacje wzajemne między sygnałem, a wzorcem. Jest to oczekiwany rezultat biorąc pod uwagę równoważność sposobu działania obu algorytmów. Oba modele wykorzystują identycznie przeprocesowane dane oraz identyczną funkcję odległości. Fundamentalna różnica polega na miejscu wykorzystania funkcji odległości. W przypadku lasów losowych dane muszą zostać przetworzone przed wykorzystaniem w modelu, w przypadku lasów kernelowych funkcja odległości jest wykorzystywana w węzłach.

Kod

Random forest

In [0]:
```
AUROC <- function(score,cls)
  mean(rank(score)[cls] - 1:sum(cls)) / sum(!cls)

res <- lapply(1:14, function(usr) {
  print(usr)
  sbjs <- ((usr-1)*80+1):(usr*80)

  # Features for correlation with pattern
  XX <- mclapply(sbjs, function(sbj) {
    apply(expand.grid(1:10, 1:7), 1, function(row) {
      chn <- row[2]
      freq <- row[1]
      apply(averaged_signals[[freq]][, chn, sbjs], 2, function(sig) {
        max(ccf(sig, averaged_signals[[freq]][, chn, sbj], lag.max=20,
 plot=F)$acf)
      })
    })
  }, mc.cores=95)

  mclapply(1:10, function(freq) {
    message(freq)

    # Features for correlation between channels.
    XX1 <- do.call(rbind, lapply(sbjs, function(sbj) {
      a <- cor(averaged_signals[[freq]][, 1:7, sbj])
      a[lower.tri(a)]
    }))

    y <- freqs[((usr-1)*80+1):(usr*80)]

    # Evaluate model.
    rf <- rFerns(cbind(XX1, as.data.frame(XX)[seq((freq-1)*560+1, freq
*560, 10)+freq-1]), as.factor(y == freq+29), ferns=5000, depth=1)
  }, mc.cores=10)
})

aucRes <- lapply(res, function(resUsr) lapply(resUsr, function(rf) {
  AUROC(rf$votes[, 2], rf$y == T)
}))
resr <- unlist(aucRes)
dim(resr) <- c(10, 14)
```

Las kernelowy

Kod lasu kernelowego był pisany na podstawie modelu flow forest (https://github.com/flowforest/flowforest (https://github.com/flowforest/flowforest)). W modelu kluczowym elementem jest funkcja makeBestSplit. Na podstawie tej funkcji szukany jest optymalny podział w węźle drzewa. Znalezienie podziałów rozróżniających obiekty różnej klasy wymaga przede wszystkim wybrania odpowiednich wzorców i odpowiedniej funkcji odległości między obiektami.

In [0]:
```
source("splitmerits.R")

bootstrap <- function(mask) {
    if (length(mask) == 1)
        return(mask)
    sample(mask, length(mask), replace = TRUE)
}

predictTree <- function(x, tree, inputMask = 1:dim(x)[1]) {
    if (length(tree) == 1)
        return(rep(tree$decision, length(inputMask)))
    # Which means: we are at leaf, return decision

    boolToLeft <- (apply(x[, tree$chn,][inputMask, , drop=FALSE], 1,
function(sig) max(ccf(sig, tree$template, lag.max=20, plot=F)$acf)) >
 tree$threshold)
    boolToRight <- !boolToLeft
    ans <- rep(NA, length(inputMask))
    ans[boolToLeft] <- predictTree(x, tree$left_subtree, inputMask[bo
olToLeft])
    ans[boolToRight] <- predictTree(x, tree$right_subtree, inputMask[
boolToRight])
    return(ans)
}

predict.aspForest <- function(object, x, ...) {
  forest <- object$forest
  votes <- rep(0, dim(x)[1])
  for(tree in forest){
    print("tree")
    fullPreds <- predictTree(x, tree, 1:dim(x)[1])
    votes <- votes + fullPreds
  }
  return(votes / length(forest))
}

aspForest <- function(x, y, ntree=200, cores=1) {
    makeBestSplit <- function(mask = 1:length(y), templateTries = 3) {
        chns <- sample(dim(x)[2], templateTries, replace=TRUE)
        templates <- sapply(1:templateTries, function(i) {
            nsbjs <- 1
            sbjs <- sample(dim(x)[1], nsbjs, replace=TRUE)
            chn <- chns[i]
            eegChn <- x[sbjs, chn, ]

            return(eegChn)
        })

        dotResults <- sapply(1:templateTries, function(i) {
            eegChn <- x[mask, chns[i], ]
            apply(eegChn, 1, function(sig) max(ccf(sig, templates[, i
], lag.max=20, plot=F)$acf))
        })

        evaluation <- colGI(dotResults, y[mask])
        iBest <- which.min(evaluation[1, ])
```

```
        bestScore <- evaluation[1, iBest]
        bestThre <- evaluation[2, iBest]
        bestSplit <- dotResults[, iBest] > bestThre
        bestParam <- iBest
        bestTemplate <- templates[, iBest]
        chn <- chns[iBest]

        return(list(
            bestTemplate = bestTemplate,
            bestScore = bestScore,
            bestParam = bestParam,
            bestThre = bestThre,
            bestSplit = bestSplit,
            chn = chns[iBest]
        ))
    }

    maxDepth = 30 #maximum depth of tree
    tree <- function(mask = 1:length(y), depth = 0) {
        # Class going downstream
        in_cls <- y[mask]
        if (depth >= maxDepth | sum(in_cls) == 0 | sum(!in_cls) == 0)
{
            # We have a leaf
            leafDecision <- mean(in_cls) > 0.5
            return(list(decision = leafDecision))
        } else {
            newSplit <- makeBestSplit(mask)
            maskLeft <- mask[newSplit$bestSplit]
            maskRight <- mask[!newSplit$bestSplit]

            return(list(
                template = newSplit$bestTemplate,
                threshold = newSplit$bestThre,
                chn = newSplit$chn,
                score = newSplit$bestScore,
                left_subtree = tree(maskLeft, depth + 1),
                right_subtree = tree(maskRight, depth + 1)
            ))
        }
    }

    forest <- list()
    oobVotes <- rep(0, length(y))
    oobTries <- rep(0, length(y))

    masks <- lapply(1:ntree, function(i) bootstrap(1:length(y)))

    fullPreds <- mclapply(1:ntree, function(i) {
      message(sprintf("Growing tree %s/%s...", i, ntree))
      forest[[i]] <- tree(masks[[i]], depth = 0)
      predictTree(x, forest[[i]])
    }, mc.cores=cores)

    for (i in 1:ntree) {
        # Extract OOB errors
```

```
        mask_oob <- !(1:length(y) %in% masks[[i]])
        oobVotes[mask_oob] <- oobVotes[mask_oob] + fullPreds[[i]][mask
_oob]
        oobTries[mask_oob] <- oobTries[mask_oob] + 1
    }
    ans <- list(forest = forest, oobVotes = oobVotes, oobTries = oobTr
ies)
    class(ans) <- "aspForest"
    return(ans)
}
```

In [0]:
```
AUROC <- function(score,cls)
  mean(rank(score)[cls] - 1:sum(cls)) / sum(!cls)

res <- lapply(1:14, function(usr) {
  x <- do.call(abind, lapply(1:10, function(freq) aperm(averaged_signa
ls[[freq]], c(3, 2, 1))[((usr-1)*80+1):(usr*80), 1:7,]), 2)
  lapply(1:10, function(freq) {
    y <- freqs[((usr-1)*80+1):(usr*80)]
    rf <- aspForest(x, y == freq+29, ntree=500, cores=95)
    AUROC(rf$oobVotes/rf$oobTries, y==freq+29)
  })
})
resr <- unlist(res)
dim(resr) <- c(10, 14)
```

Wykresy

In [0]:
```
melted_resr <- melt(resr)
melted_resr[,3] <- round(melted_resr[,3], 2)

png("AUROC_rfp5000chnfs_ccf_ind_1.png", width=1600, height=1000)
ggplot(data = melted_resr, aes(x=Var1, y=Var2, fill=value)) +
  geom_tile() + xlab("frequency") +
  scale_y_continuous("participant", 1:14) +
  scale_x_continuous("frequency", breaks=1:10, labels=as.character(30:
39)) +
  geom_text(aes(Var1, Var2, label = value), color = "black", size = 8)
 +
  scale_fill_gradient2(low = "blue", high = "red", mid = "white",
    midpoint = 0.5, limit = c(0,1), space = "Lab",
    name="AUROC") + theme(text = element_text(size=30))
dev.off()
```

Podsumowanie

Powyższa analiza ukazała jakość klasyfikacji danych przy wykorzystaniu lasów losowych i lasów kernelowych. Główna zaleta lasów losowych polega na prostocie wykorzystania wielu cech bez przeuczenia klasyfikatora. Dzięki temu w przypadku ssvep możliwe jest wykorzystanie jako cech zarówno korelacji wzajemnej między wzorcem, a sygnałem, jak i korelacji między kanałami. W przypadku szerszej przestrzeni cech klasyczne modele takie jak random forest wymagają modyfikacji. Za taką modyfikację można uznać lasy kernelowe. Dzięki liczeniu cech lokalnie na poziomie węzłów drzew możliwe jest wykorzystywanie większej liczby cech. Na tę chwilę nie znajduje to jednak zastosowania do SSVEP ze względu na brak odpowiednio dużej przestrzeni cech do wykorzystania dającej wysoki poziom klasyfikacji.

Powyższa analiza została wykonanana na podstawie danych kalibracyjnych. W przypadku danych online dochodzi wiele dodatkowych czynników mających wpływ na jakość klasyfikacji: szybkość działania klasyfikatora, dodawanie nowych przypadków do klasyfikatora w celu poprawy jakości klasyfikacji, długość czasu podejmowania decyzji przez uczestnika, wybór częstości migania oraz ich ilości. Wszystkie te czynniki są bardzo istotne w przypadku projektowania interfejsu i powodują potrzebę testowania klasyfikatora na danych online.

Podział obowiązków

Katarzyna Filipiuk - zarządzanie projektem (trello.com, cocalc.com); dostęp do danych; ekstrakcja i przygotowanie danych; rozdziały Wstęp i Ekstrakcja danych;
Krzysztof Piwoński - analiza danych z wykorzystaniem lasów losowych i lasów kernelowych;
Urszula Romaniuk - zarządzanie projektem (trello.com, cocalc.com); dostęp do danych; ekstrakcja i przygotowanie danych; rozdziały Wstęp i Ekstrakcja danych;
Izabela Szopa - zarządzanie projektem (trello.com, cocalc.com); dostęp do danych; ekstrakcja i przygotowanie danych; rozdziały Wstęp i Ekstrakcja danych;