Term work 2

```
install.packages("KernelKnn")
data(ionosphere, package = 'KernelKnn')
apply(ionosphere, 2, function(x) length(unique(x)))

# the second column will be removed as it has a single unique value
ionosphere = ionosphere[, -2]

#Scale the data since the output depends on the distance calculations
X = scale(ionosphere[, -ncol(ionosphere)])
y = ionosphere[, ncol(ionosphere)]

# labels should be numeric and begin from 1 since classification is used
y = c(1:length(unique(y)))[ match(ionosphere$class, sort(unique(ionosphere$class))) ]


# random split in train-test and test set.
spl_train = sample(1:length(y), round(length(y) * 0.75))

#The elements of setdiff(x,y) are those elements in length(y) but not in spl_train
spl_test = setdiff(1:length(y), spl_train)
str(spl_train)
str(spl_test)



# evaluation metric
acc = function (y_true, preds) {

  out = table(y_true, max.col(preds, ties.method = "random"))

  #A key metric to start with is the overall classification accuracy.
  #It is defined as the fraction of instances that are correctly classified.
  acc = sum(diag(out))/sum(out)

  acc
}
```

#A simple k-nearest-neighbors can be run with weights_function = NULL and the parameter 'regression' should be set to FALSE. In classification the Levels parameter takes the unique values of the response variable,

```
library(KernelKnn)
```

```
preds_TEST = KernelKnn(X[spl_train, ], TEST_data = X[spl_test, ], y[spl_train], k = 5 ,
                method = 'euclidean', weights_function = NULL, regression = F,
                Levels = unique(y))
head(preds_TEST)



#There are two ways to use a kernel in the KernelKnn function. The first option is to choose one of the existing kernels (uniform, triangular, epanechnikov, biweight, triweight, tricube, gaussian, cosine, logistic, silverman, inverse, gaussianSimple, exponential). Here, I use the canberra metric and the tricube kernel because they give optimal results
preds_TEST_tric = KernelKnn(X[spl_train, ], TEST_data = X[spl_test, ], y[spl_train], k = 10 ,
                   method = 'canberra', weights_function = 'tricube', regression = F,
                   Levels = unique(y))
head(preds_TEST_tric)



#The second option is to give a self defined kernel function. Here, I'll pick the density function of the normal distribution with mean = 0.0 and standard deviation = 1.0
norm_kernel = function(W) {

  W = dnorm(W, mean = 0, sd = 1.0)

  W = W / rowSums(W)

  return(W)
}
preds_TEST_norm = KernelKnn(X[spl_train, ], TEST_data = X[spl_test, ], y[spl_train], k = 10 ,
                   method = 'canberra', weights_function = norm_kernel, regression = F,
                   Levels = unique(y))
head(preds_TEST_norm)


#I'll use the KernelKnnCV function to calculate the accuracy using 5-fold cross-validation for the previous mentioned parameter pairs,
fit_cv_pair1 = KernelKnnCV(X, y, k = 10 , folds = 5, method = 'canberra',

                   weights_function = 'tricube', regression = F,

                   Levels = unique(y), threads = 5)
str(fit_cv_pair1)
fit_cv_pair2 = KernelKnnCV(X, y, k = 9 , folds = 5,method = 'canberra',
```

```
                     weights_function = 'epanechnikov', regression = F,

                     Levels = unique(y), threads = 5)
str(fit_cv_pair2)




#Each cross-validated object returns a list of length 2
acc_pair1 = unlist(lapply(1:length(fit_cv_pair1$preds),

                    function(x) acc(y[fit_cv_pair1$folds[[x]]],

                                fit_cv_pair1$preds[[x]])))
acc_pair1
cat('accurcay for params_pair1 is :', mean(acc_pair1), '\n')
acc_pair2 = unlist(lapply(1:length(fit_cv_pair2$preds),

                    function(x) acc(y[fit_cv_pair2$folds[[x]]],

                                fit_cv_pair2$preds[[x]])))
acc_pair2
cat('accuracy for params_pair2 is :', mean(acc_pair2), '\n')
```