Universita Commerciale Luigi Bocconi

Individual Assignment

Course Code	30416
Course Name	BIG DATA AND DATABASES
Title of Assignment	Individual Assignment
Student Name	Deniz Oezdemir - 3124739
Professor	LUCA MOLTENI, DANIELE TONINI, ALESSANDRO REZZANI
Assignment Due Date	December 12th, 2019
Date of Submission	December 12th , 2019
Number of pages (excluding Cover Page):	27

Data Analysis Project on Human Activity Recognition

Table of Contents

1 Goal of the Analysis 3
2 First Description of the Dataset 3
2.1 Experimental Setup 3
3 Statistical Description 4
3.1 Raw Data Description 4
3.2 Data Visualization 5
4 Transforming the Data 6
4.1 Standardization and Principal Component Analysis 6
4.2 Feature Selection: 7
5 Exploring Algorithm Solutions 8
5.1 Algorithm Explanation 8
5.2 Algorithm Evaluation 9
6 Experimental Phase: Artificial Neural Networks (ANN) 10
6.1 Theoretical Intuition of ANNs 10
6.2 Defining the Optimal ANN for the given Dataset: Hyperparameter Tuning 11
6.2.1 Parameters under Consideration 11
6.2.2 Procedures of Hyperparamter Tuning 13
6.2.3 Actual Coding 14
6.3 Visualizing the ANN 16
7 Initializing the Models with KNIME 17
7.1 Importing the Data 17
7.2 Principal Component Analysis 17
7.3 Algorithm Creation 19
7.3.1 The Artificial Neural Network 20
7.3.2 The Logistic Regression 24
7.4 Model Evaluation 25
7.5 Implications for Data Scientists and Mangers 27
7.5.1 Interpretation of the Results and Implications for the Data Scientist 27
7.6 Implications for Managers 28
8 Works Cited 28

1 Goal of the Analysis

Human Activity Recognition is a central topic for several different contexts and instiutions. One can think about the Government's interest in enhancing surveillance in urban areas, business entities trying monitor and optimize worker's behavior, as well as in healthcare and many other areas.

The goal of this study is to enter in the active research area of Human Activity Recognition, and provide yet another example of how the techniques of Machine and Deep Learning Algorithms can be used to tackle a problem that previously needed human operators.

A second goal of this study will be to show through a lengthy experiment that it can in specific cases be rewarding to choose models with lower complexity without losing a lot of predictive capability, but gaining a lot in the sphere of computational speed.

2 First Description of the Dataset

2.1 Experimental Setup

The Experiment was performed by 30 participants, each performing the following six activates while carrying a waist-mounted smartphone with embedded inertial sensors:
WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING
Using its embedded accelerometer and gyroscope, the following data were captured:
3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz.

The sensor signals were then sampled in sliding windows of 2.56 seconds. The sensor signal was decomposed into gravitational and body motion components with a low-pass filter.
For each window, a vector of features was obtained.

The experiment has been video-recorded and afterwards labeled according to the six activities.

We therefore end up with an 562-feature vector with time and frequency domain variables as well as its activity label.

3 Statistical Description

3.1 Raw Data Description

```
              ###having a look at the Raw Data###
###dimension of data: (7352,563)
print(dataset_train.shape)
###head of data:
print(dataset_train.head(20))
###statistical description: all attributes have the same range -1 to 1 (apart from subject)
print(dataset_train.describe())
### Class Balance: from 986 to 1407
print(dataset_train.groupby('Activity').size())
```

Running the above script, we see that the data has 7352 observations in the training set, and 563 attributes including the output attribute Activity.

```
(7352, 563)
```

Analyzing the Descriptive Statistics of the different variables, it becomes clear that there are no missing values and the value-range lies between -1 and +1, as can be seen in the example of three attributes below:

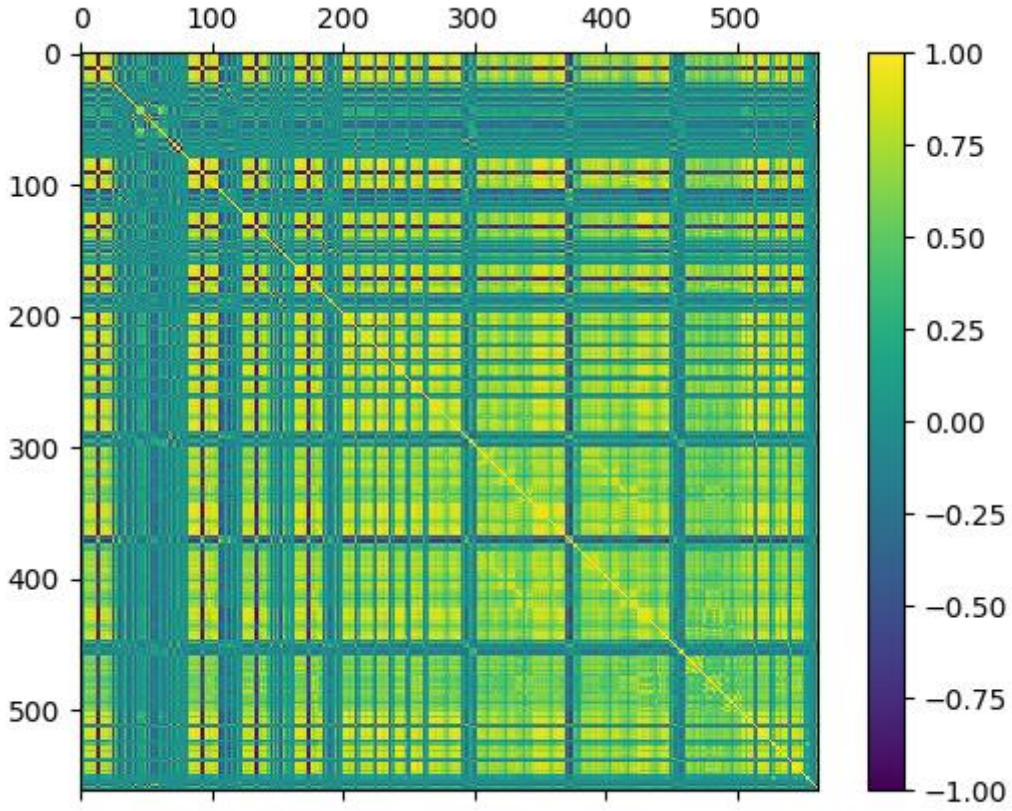
```
       tBodyAcc-mean()-X  tBodyAcc-mean()-Y  tBodyAcc-mean()-Z  \
count        7352.000000        7352.000000        7352.000000
mean            0.274488          -0.017695          -0.109141
std             0.070261           0.040811           0.056635
min            -1.000000          -1.000000          -1.000000
25%             0.262975          -0.024863          -0.120993
50%             0.277193          -0.017219          -0.108676
75%             0.288461          -0.010783          -0.097794
max             1.000000           1.000000           1.000000
```

We also see that the distribution between the different classes of the output variable “Activity” is quite balanced, with values between 986 and 1407, which is a ratio around 5 to 7.

```
Activity
LAYING                1407
SITTING               1286
STANDING              1374
WALKING               1226
WALKING_DOWNSTAIRS     986
WALKING_UPSTAIRS      1073
dtype: int64
```

3.2 Data Visualization

```
                ###Data Visualization###
# correlation matrix
fig = pyplot.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(dataset_train.corr(), vmin=-1, vmax=1, interpolation='none')
fig.colorbar(cax)
pyplot.show()
```



Looking at the Correlation Matrix we see that there is a lot of Multicollinearity between the attributes. This points means that we might profit from running a Principal Component Analysis in order to reduce the Dimensionality of the dataset and therefore get rid of correlated attributes.

4 Transforming the Data

4.1 Standardization and Principal Component Analysis

In order to get rid of the correlation between the attributes, it may be rewarding to use a technique in order to reduce dimensionality of the dataset. Therefore we will use the Principal Component Analysis, which basically transforms our linearly correlated dataset into a set of uncorrelated features, so called Principal Components. This is done by an eigenvalue decomposition of the correlation matrix, and only those eigenvalues which explain more than one variable (with values greater than 1) are held. It is important that the data are standardized, in order not to impact the PCA by differences in scales because we are scaling the covariance between every pair of variables by the product of the standard deviations of each pair of variables.

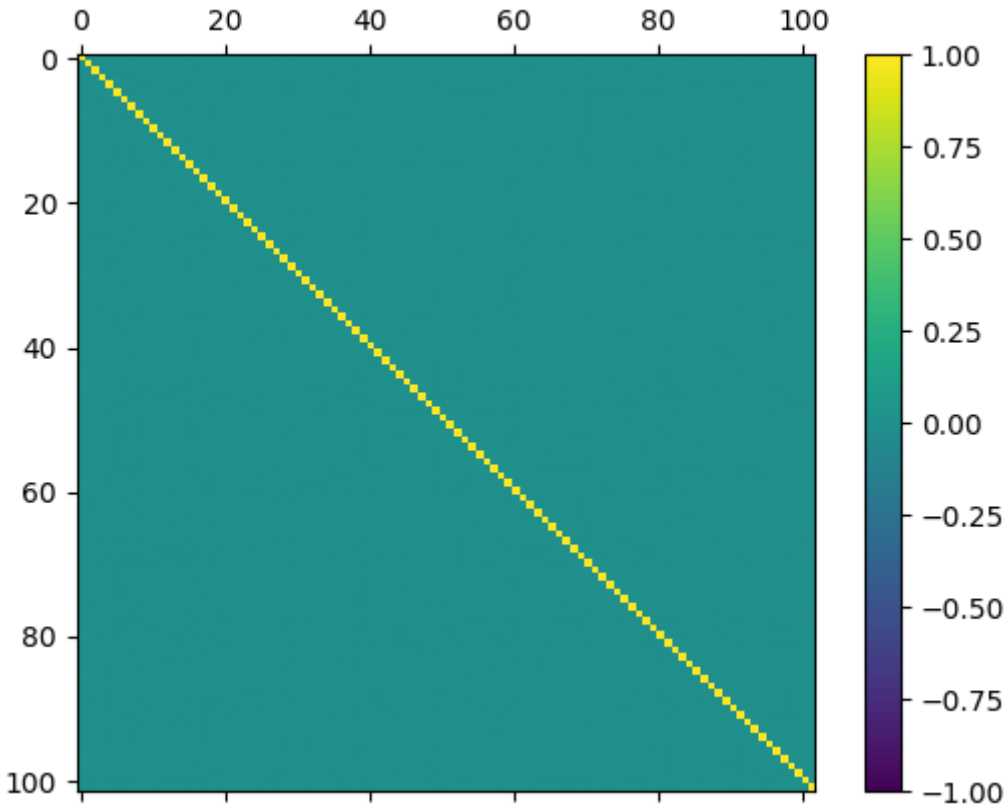
Note: It is very important that we do not let the standard-scaler see the testing data, since that would be an interference between training and testing and would harm the integrity of the data analytics process.

```
#Train/Test Split

array_train = dataset_train.values
X_train = array_train[:,0:562]
Y_train = array_train[:,562]

array_test = dataset_test.values
X_test = array_test[:,0:562]
Y_test = array_test[:,562]

#print(Y_train[0:5])

##getting rid of Multicollinearity in the Dataset

#standardize the attributes (fit the scaler only on train data!)
scaler = StandardScaler()
# Fit on training set only.
scaler.fit(X_train)
# Apply transform to both the training set and the test set.
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Make an instance of the PCA Model, set on .95 of explained variance
pca = PCA(.95)
#Fit on training set only
pca.fit(X_train)
#apply transformation to the two datasets
X_train = pca.transform(X_train)
X_test = pca.transform(X_test)


#Check what happened
print(X_train.shape)
print(X_test.shape)
```

In our example, we want to retain at least 95% of the variance of our original dataset, in order not to harm our predictive capabilities later on.

```
(7352, 102)
(2947, 102)
```

As we can easily see, we reduced our attributes from 562 to 102, which corresponds to a reduction of 82%.



As we can observe after applying once again a Correlation Matrix Visualization, there are no correlated features anymore.

4.2 Feature Selection:

It was considered at this point to use further feature selection on the grounds of a tree-based feature importance measurement.
However, since decision trees do not seem to perform well on the given problem set (having the lowest accuracy score in cross-validation as we will see in the next step), it might be misleading to remove features a decision tree does not use for making splits, since the dynamics of tree-based algorithms and their split decisions are not tailored for this problem.
Also a selection based on correlation between attributes and target variable is not possible, since the target is a categorical variable.
Therefore, we stick to the PCA feature extraction rather than applying additional feature selection. However, in KNIME we will see that the total variance captured by the PCA can be slightly adjusted downwards without making enormous accuracy losses.

5 Exploring Algorithm Solutions

5.1 Algorithm Explanation

In order to get a first feeling for which classification algorithms may produce good outcomes, the following models are considered:

- Logistic Regression (LR)
  - LR is a special case of linear Regression in which the dependent variable is the log of odds.
- Linear Discriminant Analysis (LDA)
  - LDA seperates datapoints by moving them into another feature space
- K-nearest neighbor classifier (KNN)
  - KNN is an algorithm that segments the datapoints into k segments, and lets the nearest neighbors in the segment of the to predicted attribute vote on its class.
- Classification And Regression Tree (CART)
  - CART is a decision tree that makes split-decision according to the values of input variables, in order to find the right class for the output variable
- Native Bayes (NB)
  - NB uses Bayesian Probability to predict the outcome class
- Support Vector Machine (SVM)
  - SVM creates a line or a hyperplane that divides the datapoint into classes

```
#Create Algorithms

models = []
models.append(('LR', LogisticRegression(solver='lbfgs')))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='scale')))
```

5.2 Algorithm Evaluation

```
#evaluate models: LDA an LR have the highest accuracy
results = []
names = []
scoring = 'accuracy'
n_folds = 10
seed= 7
for name, model in models:
    kfold = KFold(n_splits = n_folds, random_state = seed)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s : %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

In order to assess whether the algorithms do a good job on the training data, there are different ways to test the accuracy. One common way is to create a holdout from the training set, a so-called “validation set”, which is used to evaluate the performance of your algorithms after it has been trained on the remaining training data. This however, reduces the data we need to train the models, therefore could lead to underfitting.
To hinder that, we will use K-Cross-Validation. Here we create K subsamples of the training set, choose randomly one holdout, and train the data with the remaining k-1 subsamples. We repeat this process several times and get a reliable accuracy score.

```
LR : 0.914991 (0.049538)
LDA : 0.910096 (0.045248)
KNN : 0.869427 (0.029850)
CART : 0.737896 (0.042563)
NB : 0.781006 (0.037113)
SVM : 0.917574 (0.037415)
```

We see that Logistic Regression seems to perform pretty well on the dataset with an accuracy score of 91%, which already implies that the dataset is linearly separable.

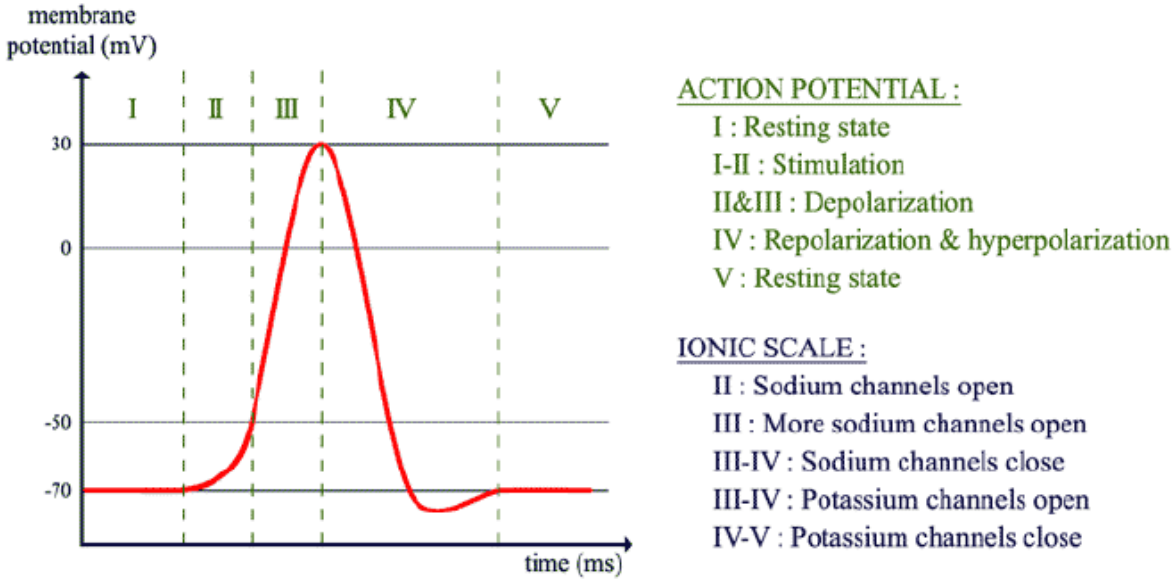
However, since Artificial Intelligence and Big Data are currently going through a hype-cycle which is driven by the advances of Deep Learning, it might be an interesting experiment to test whether an optimally trained Artificial Neural Network will do an even better job on this problem than simple linear models do.

6 Experimental Phase: Artificial Neural Networks (ANN)

6.1 Theoretical Intuition of ANNs

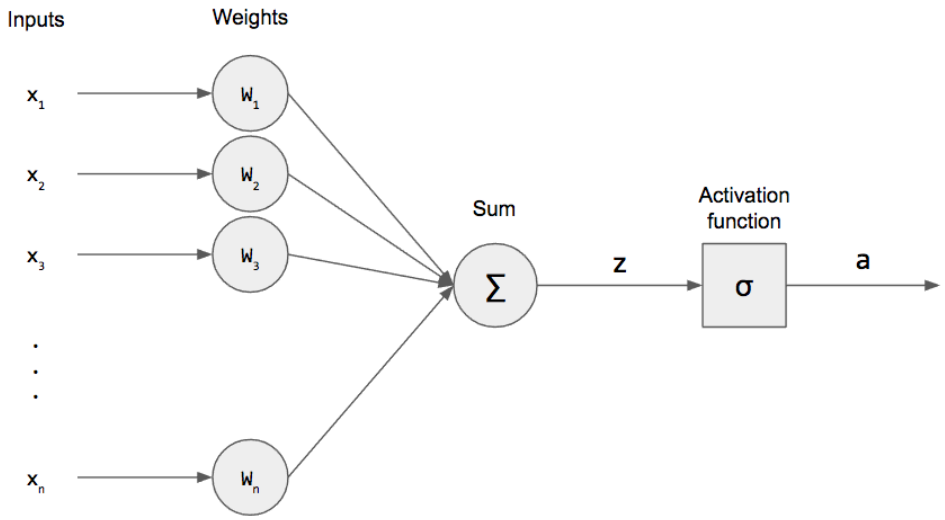
Artificial Neural Networks (ANN), since their first introduction in 1943 (McCulloch und Pitts), have evolved massively from what used to be algorithms created via electronic circuits. After nearly 80 years of relevant research, interest in such networks have sparked once again over academic boundaries into the mainstream, in an industry that has already experienced several hype-cycles (Gartner).

In essence, ANNs are mathematical descriptions of what we know about the functionality of human neurons. The Artificial Neurons (or Perceptrons) simulate the firing behavior of a biological neuron. it may be useful to explain it with the Hodgkin-Huxley model, which describes how action potentials in neurons are activated through the opening and closing of the sodium and potassium channels that are dependent on the voltage of the cell, which in turn is in a nonlinear relationship with ionic currents across the neuronal cell membrane. A simple intuition for that is that, the model describes via ion channels how electrical stimulation at the input level is related to the membrane output voltage, which basically is the action potential in form of a spike-signal.



(Corson)

In comparison to that, the Perceptron mathematically describes this activation of the neutron by the following threshold idea:



The inputs simulate the stimuli via synapses, wheras the weights attached to the inputs symbolize synaptic density. The weighted inputs are simply summed up, and in its most simple form they activate the perceptron if above a certain threshold.

A whole ANN is meant so simulate so called Cell Assemblies. These CAs are described as populations of neurons which have certain common characteristics. As such, they can be clustered in a common structure. Those commonalities are described, in the so called “Hebbian Theory” (Hebb), whose explanation is well beyond the scope of this paper (if interested in this type of cognitive science artificial intelligence research, read (Pulvermüller) or wait for 2020 paper (Oezdemir)). The layers and parameters of ANNs will be described as we go along with the Experiment.
ANNs can be used to model linear, as well as non-linear relationships

6.2 Defining the Optimal ANN for the given Dataset: Hyperparameter Tuning

6.2.1 Parameters under Consideration

In order to find the optimal network for the given problem, the parameters who constitute and influence the learning process of the network, the so called “Hyperparameters”, need to be configured in an ideal way. For a feedforward neural network used for a classification on structured data, the following parameters will be considered:

- Number of Hidden Layers:
  - Those are the layers containing neurons between the input and the output layer.
  - Even though the Universal approximation theorem states that a single hidden layer can approximate any continuous function (Cybenko), it is today known that the width of such networks has to be exponentially large and that we can lower drastically computational cost by adding extra layers.
  - Therefore, we will decide between two and three hidden layers.

- Number of Neurons per Layer:
  - There are only heuristic rules for the choice of neurons per layer, such as staying between 1 and the number of inputs.
  - Therefore we choose neurons for all three layers in the space { 10, 20, 40, 60, 100}
- Dropout Rate:
  - When using neural networks, it can happen that the neurons in the model co-adapt to correct mistakes made by prior layers and neurons, therefore capturing all the statistical noise in the training data, therefore leading to over-fitting
  - In order to hinter this co-adaption of neurons, regularization techniques are used to ignore randomly certain neurons in the layers, in order to force the remaining neurons to concentrate on the inputs rather than on reducing the errors of other neurons.
  - We choose a space of { 0.25,0.5,0.75} for all three layers.
- Activation Function
  - The activation function, as described in above, defines how a neuron deals with the weighted inputs it gets.
  - We will consider the following functions: 'softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear'
- Kernel Initializer:
  - This basically describes which statistical distribution is used for initializing the weights while setting up the network
  - We will set that to uniform distribution
  - Kernel Weight Constraint:
    - This is a regularization parameter that penalizes high weights with a loss function
    - We set it to 3, since that is known to work well for most ANNs
- Optimizer Function:
  - The Optimizer function describes the function that will be used to tune the weights of the neurons in such a way that the Objective Function (or the Error Function) is minimized.
  - We will use Stochastic Gradient Descent for that, which is basically an algorithm to find the local minimum of the error function.
  - However, in this capacitiy we need to tune further parameters connected to the Optimizer:
    - Batch Size:
      - Basically determines how many observations to go through before updating the weights
      - We use a space of { 20,60,100}
    - Epochs:
      - The number of times the algorithm will go through the whole dataset
      - We use a space of { 10, 50, 100}

- Learning Rate:
  - The step size the Stochastic Gradient Descent will go down in the direction of the negative gradient with every iteration.
  - We test { 0.001, 0.01, 0.1}
- Momentum:
  - In simple terms, helps accelerate the gradient vectors in the right direction, therefore leading to quicker convergence
  - We consider a space of { 0.0, 0.2, 0.4, 0.6, 0.8}

6.2.2 Procedures of Hyperparamter Tuning

There are several ways to "tune" these parameters to a value that optimizes the predictive capabilities of the ANN:

1.) Grid Search:

Here we set up the search space as described above, and go through every possible combination of the different values, train the model and then test it in order to find out the configuration with the highest performance metrics.
In our case, that would sum up to above 1 million possible combinations, meaning more than a million networks to train and then to cross validate every one several times.
Obviously, this cannot be done in a reasonable amount of time on a computer not built for such computations. Therefore, the choice is either to rent a cloud service such as AWS, or to go for another procedure.

2.) Random Search

This is another greedy algorithm, just instead of being exhaustive as Grid Search, it chooses combinations by random.
This will yield good results when tuning a limited number of hyperparameters, which is not the case in our experiment

3.) Bayesian Optimization

Bayesian Optimization differs from Grid Search and Random Search in the aspect that it updates its knowledge of the parameters with each configuration and memorizes those settings that actually increase the accuracy of the model, therefore it limits the search space it searches and deliberately ignores those settings that have proven to bring nothing new to the table.
It does so by the help of a surrogate and selection function. In order to avoid firing the objective function (or loss function) for every configuration of parameters, which means training the whole model, Bayesian optimizing uses a simplified approximiation of the objective function which it tries to minimize, the so called surrogate function. A typical surrogate function is the Tree Parzen Estimator (TPE), which maps hyperparmaters to a probability of a score on the objective function. It does so by Bayesian updating of the prior distribution with every new set of data available. With each iteration the surrogate becomes a better predictor of the objective function.
The hyperparmeters tested are, as previously stated, limited, and that is done by applying a criterion to the surrogate function, the so-called selection function.
By doing that through several iterations, we can zero in on the perfect configuration, without having to test every possible combination of hyperparameters.

6.2.3 Actual Coding

In code, that looks as follows:

```
space = {'choice': hp.choice('num_layers', [{'layers':'two',},{'layers':'three',
                                             'units3': hp.choice('units3', [10, 20, 40, 60, 100]),
                                             'dropout3': hp.choice('dropout3', [0.25,0.5,0.75])}]),
    'units1': hp.choice('units1', [10, 20, 40, 60, 100]), #number of neurons in the first layer
    'units2': hp.choice('units2', [10, 20, 40, 60, 100]), # number of neurons in the second layer

    'dropout1': hp.choice('dropout1', [0.25,0.5,0.75]),
    'dropout2': hp.choice('dropout2', [0.25,0.5,0.75]),

    'batch_size' : hp.choice('batch_size', [20,60,100]),
    'nb_epoch' : hp.choice('nb_epoch', [10, 50, 100]),
    'learn_rate' : hp.choice('learn_rate', [0.001, 0.01, 0.1]),
    'momentum' : hp.choice('momentum', [0.0, 0.2, 0.4, 0.6, 0.8]),
    'activation': hp.choice('activation', ['softmax', 'softplus',
                                           'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']),
    'optimizer': 'SGD',
    'init_mode':'uniform',
    'weight_constraint': 3
    }
```

Here we create the search space.

```
def train_fcn(features, labels, params):

    model = Sequential()
    model.add(Dense(output_dim=params['units1'], input_dim = X_train.shape[1],
                    activation=params['activation'], kernel_initializer = params['init_mode']))
    model.add(Dropout(params['dropout1']))
    model.add(BatchNormalization())

    model.add(Dense(output_dim=params['units2'], activation=params['activation'],
                    kernel_initializer = params['init_mode']))
    model.add(Dropout(params['dropout2']))
    model.add(BatchNormalization())

    if params['choice']['layers']== 'three':
        model.add(Dense(output_dim=params['choice']['units3'],
                        activation=params['activation'], kernel_initializer = params['init_mode']))
        model.add(Dropout(params['choice']['dropout3']))
        model.add(BatchNormalization())


    model.add(Dense(6, kernel_initializer=params['init_mode'], activation='softmax'))
    optimizer = SGD(lr=params['learn_rate'], momentum=params['momentum'])
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    model.fit(features, labels, epochs=params['nb_epoch'], batch_size=params['batch_size'], verbose=0)
    return model
```

After that we connect our ANN (built using the library KERAS) with the search space.
You may notice that the ouput layer actually has 6 neurons, which is a result of the One-Hot Enconding of the Target Variable Activity into a dummy variable for each of its classes, since the ANN can only output a binary value for each neuron.

```
#one hot encoding for the labels/Y variables
encoder = LabelEncoder()
encoder.fit(Y_train)
Y_train = encoder.transform(Y_train)
# convert integers to dummy variables (i.e. one hot encoded)
Y_train = to_categorical(Y_train)

#one hot encoding for the labels/Y variables
encoder = LabelEncoder()
encoder.fit(Y_test)
Y_test = encoder.transform(Y_test)
# convert integers to dummy variables (i.e. one hot encoded)
Y_test = to_categorical(Y_test)
```

The one hot enconding is done here. The label Enconder needs to be separately handeled from the testing data since any interference between testing data and model needs to be hindered.

```
def test_fcn(model, features, labels):
  test_accuracy = model.evaluate(features, labels)
  return test_accuracy

X_train, X_val, Y_train, Y_val = model_selection.train_test_split(X_train, Y_train, stratify=Y_train)

def hyperopt_fcn(params):
  model = train_fcn(X_train, Y_train, params)
  test_acc = test_fcn(model, X_val, Y_val)
  K.clear_session()
  return {'loss': -test_acc[1], 'status': STATUS_OK}
```

Then we create the objective function

```
bayes_trials = Trials()

best = fmin(fn = hyperopt_fcn, space = space, algo = tpe.suggest,
            max_evals = 50, trials = bayes_trials, rstate = np.random.RandomState(50))

print (best)
print (bayes_trials.best_trial)
```

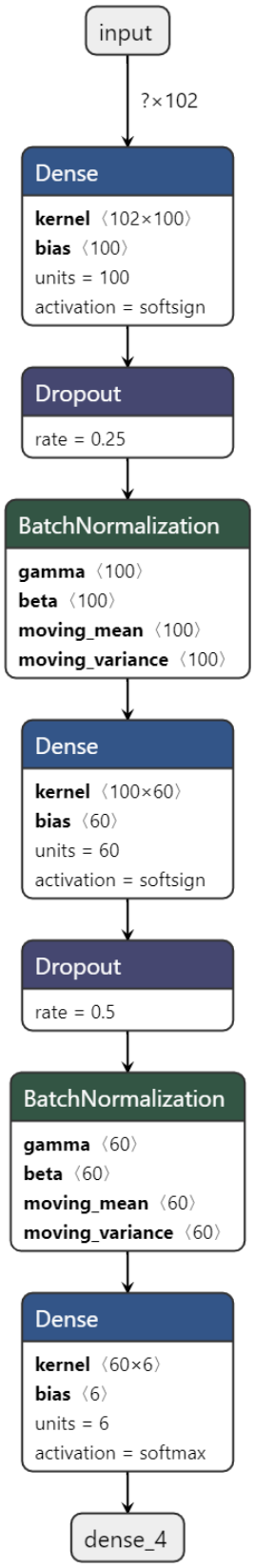
Ultimately we use the HyperOpt library to perform the optimization as described above and to print the best set of output variables.

```
Out[34]:
{'activation': 'softsign',
 'batch_size': 100,
 'choice': {'layers': 'two'},
 'dropout1': 0.25,
 'dropout2': 0.5,
 'init_mode': 'uniform',
 'learn_rate': 0.1,
 'momentum': 0.8,
 'nb_epoch': 100,
 'optimizer': 'SGD',
 'units1': 100,
 'units2': 60,
 'weight_constraint': 3}
```

We end up with this optimal set of Hyperparameters found through Bayesian Optimizing, specific for our dataset at hand.

6.3 Visualizing the ANN

In order to get a quick overview over the network without having to go through the code, the network we found optimal is visualized with the help of the software NETRON, which is a viewer for neural network, deep learning and machine learning models.

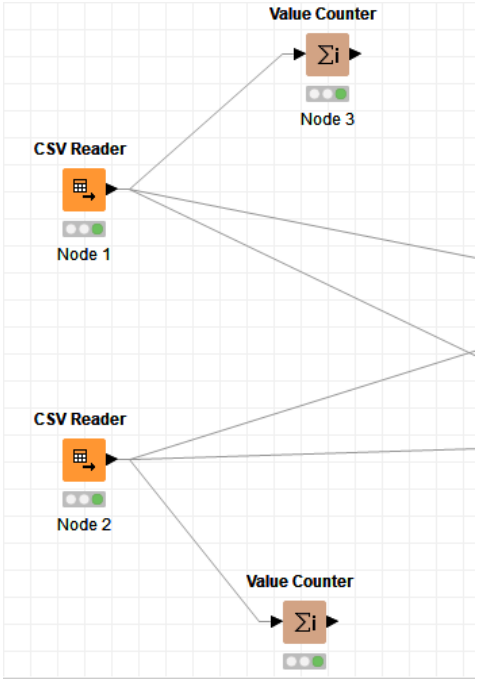


Here it might be a good idea to explain the Batch Normalization Layer:
Since we use batches (subsamples) to train the ANN in every epoch, the distribution of the input data changes from batch to batch. That means that the weights have to adjust for that, which feeds into the whole network and slows training down.

In order to minimize this effect called “internal covariate shift” we use batch normalization to normalize the inputs to each layer.

7 Initializing the Models with KNIME

7.1 Importing the Data

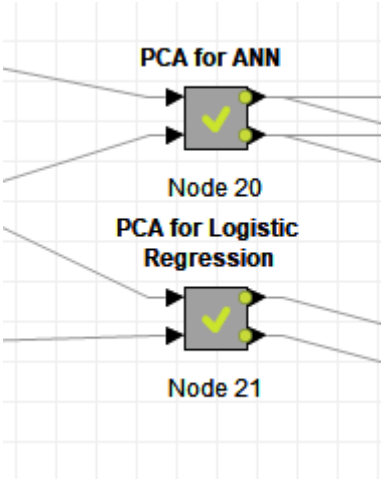


Row ID	count
LAYING	1407
SITTING	1286
STANDING	1374
WALKING	1226
WALKING_DO...	986
WALKING_UP...	1073

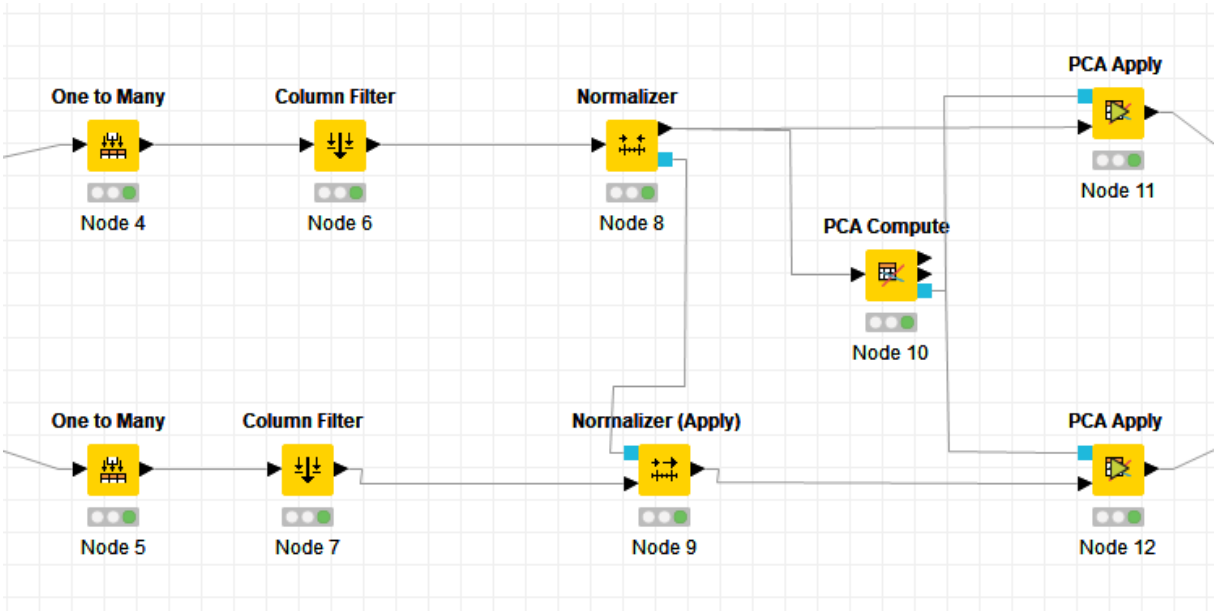
Row ID	count
LAYING	537
SITTING	491
STANDING	532
WALKING	496
WALKING_DO...	420
WALKING_UP...	471

Here we see that the classes are pretty much balanced in both the training and the testing data set.

7.2 Principal Component Analysis

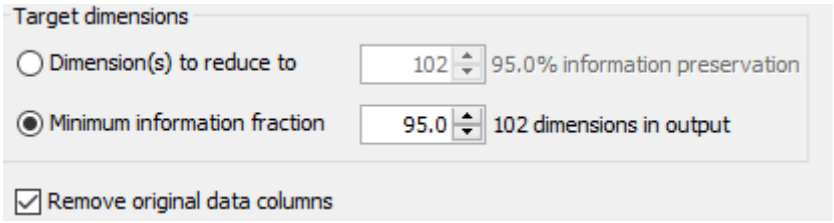


Here we created two meta nodes. One which captures 95% of the Variance of the dataset, as already seen in the Python code, which will be used to compare directly the ANN with a linear model, in this case the logistic regression.
However we also create another PCA note where we scale down a bit the Variance we want to capture, to see whether or not that has enormous effects on the capability of prediction of our Logistic Regression model in a second instance.

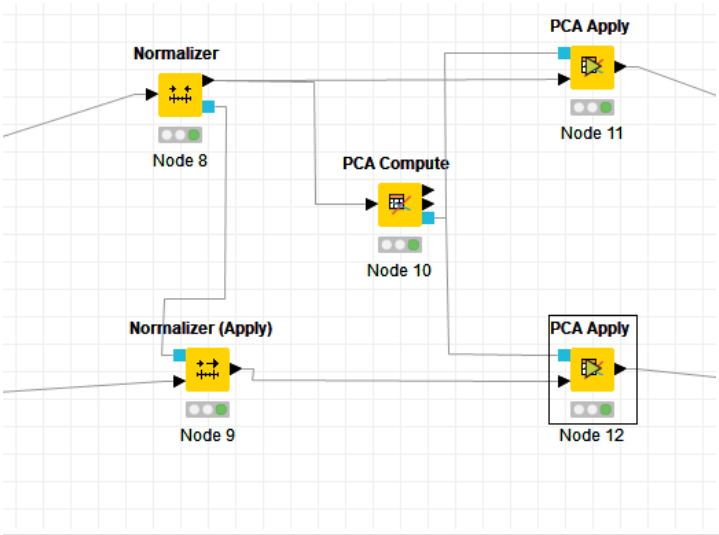


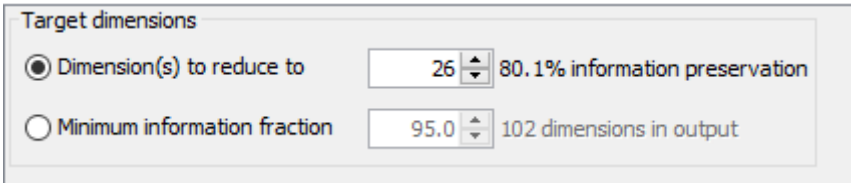
Inside the PCA node we see first a one-hot encoding done via the “One to Many” node, then we use the Normalizer node to perform Gaussian Normalization (the right term is standardization, so this can be confusing).

In the actual PCA nodes we respect the partition between training and testing, so we only fit the PCA on the training set and then transform the test set.



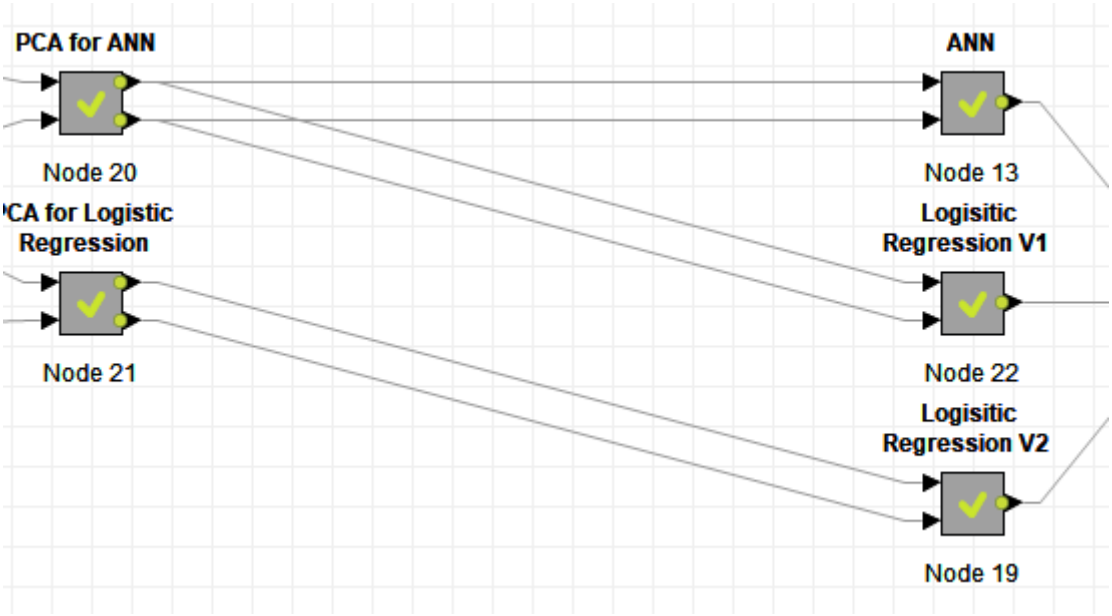
We use information fraction of 95%.





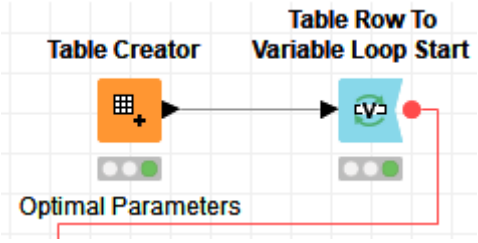
Inside the second node we have a shorter picture, since Logistic Regression does not demand one-hot encoding. For the PCA, we use information fraction of 80% and end up with 26 principal components. We will see how this affects our model training.

7.3 Algorithm Creation



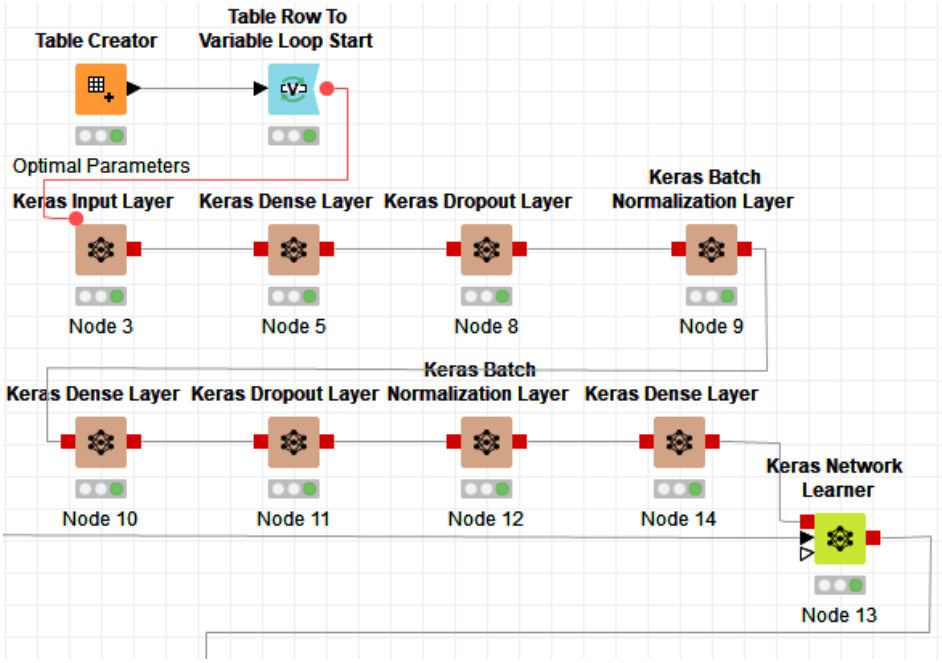
We then link the PCAs to three models.

7.3.1 The Artificial Neural Network

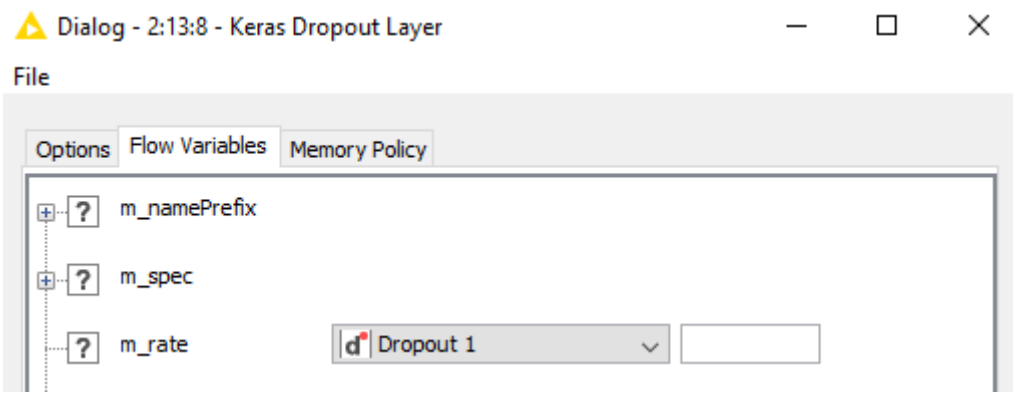


Name	Value
currentIteration	0
maxIterations	1
Activation	Softsign
Batch Size	100
Dropout 1	0.25
Droput 2	0.5
Init Mode	Glorot Uniform Normalizer
Learning Rate	0.1
Momentum	0.8
Epochs	100
Optimizer	SGD
Units 1	100
Units 2	60
Weight Constraint	3.0

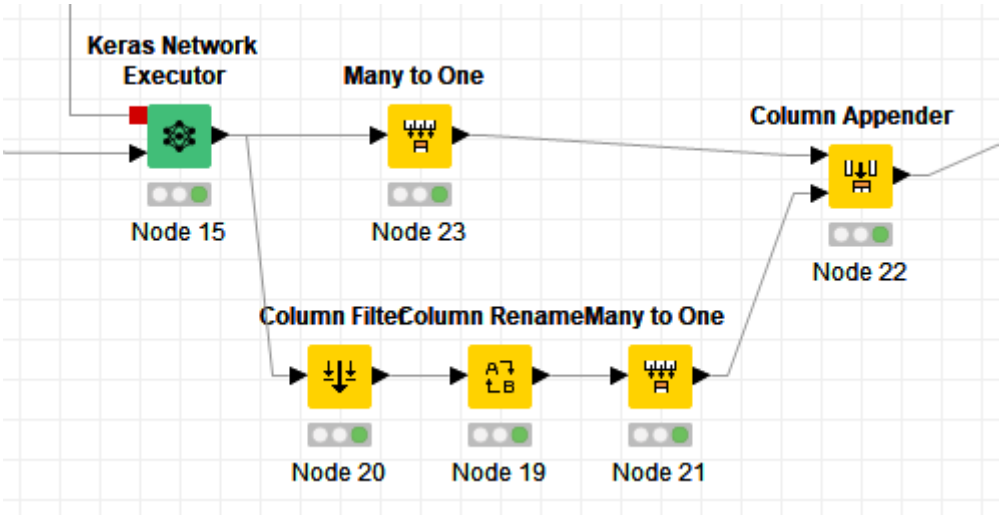
Inside the ANN node we first create flow variables according to the Optimal Hyperparameters we found in the Experimentation Phase of this project.
These flow variables are then used to determine the hyperparameters of the nodes of the ANN.



The network is built with the KNIME KERAS extension nodes, so it is directly comparable with the visualization of the Network we have shown earlier.



As an example of how the flow variables are connected within the nodes, we can look into the configuration of the first Dropout layer.

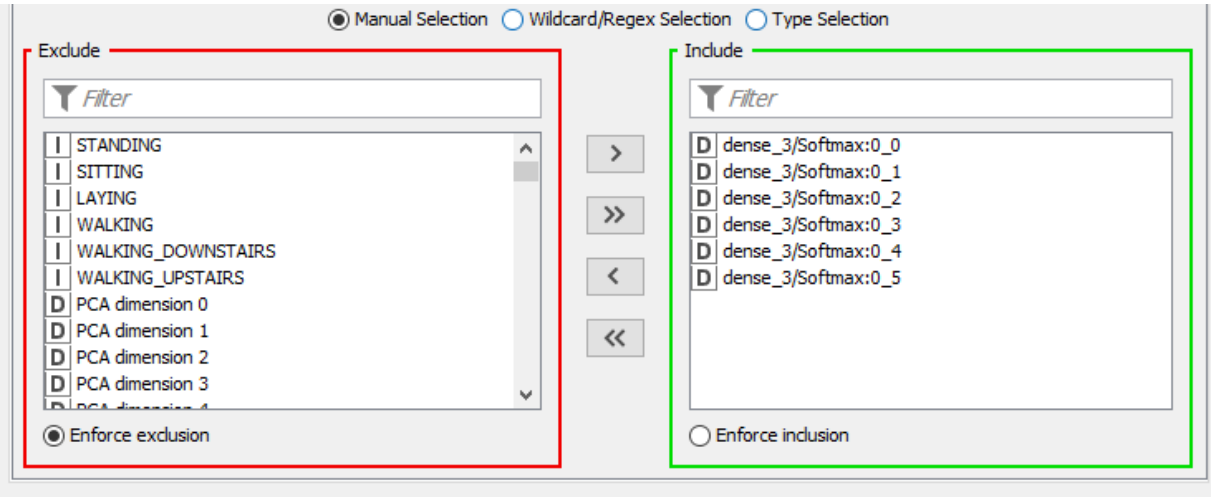


After the network has been trained, certain data transformations are required:

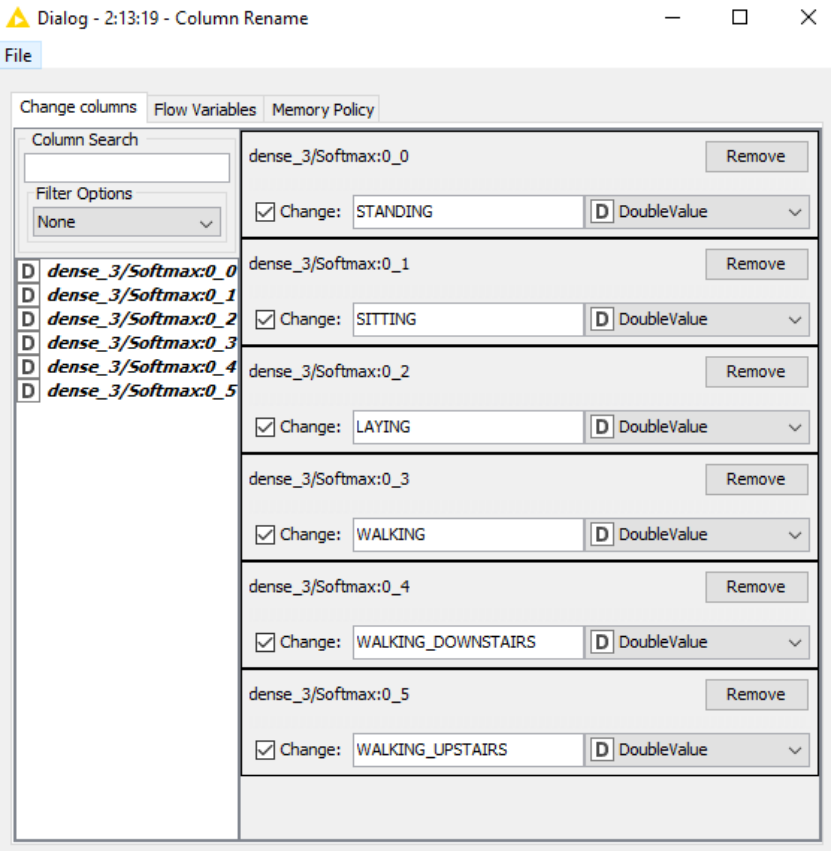
1.) We have to undo the one-hot encoding and create a single class for the Activity again, so that we later can put Actual Class next to Prediction.
2.) we have to consider certain aspects of the output of an Artificial Neural Network built with KERAS in KNIME:

D dense_...	D dense_...	D dense_...	D dense_...	D dense_...	D dense_...
1	0	0	0	0	0
0.995	0.005	0	0	0	0
1	0	0	0	0	0
0.999	0.001	0	0	0	0
1	0	0	0	0	0
1	0	0	0	0	0
0.999	0.001	0	0	0	0
0.965	0.035	0	0	0	0
0.989	0.011	0	0	0	0

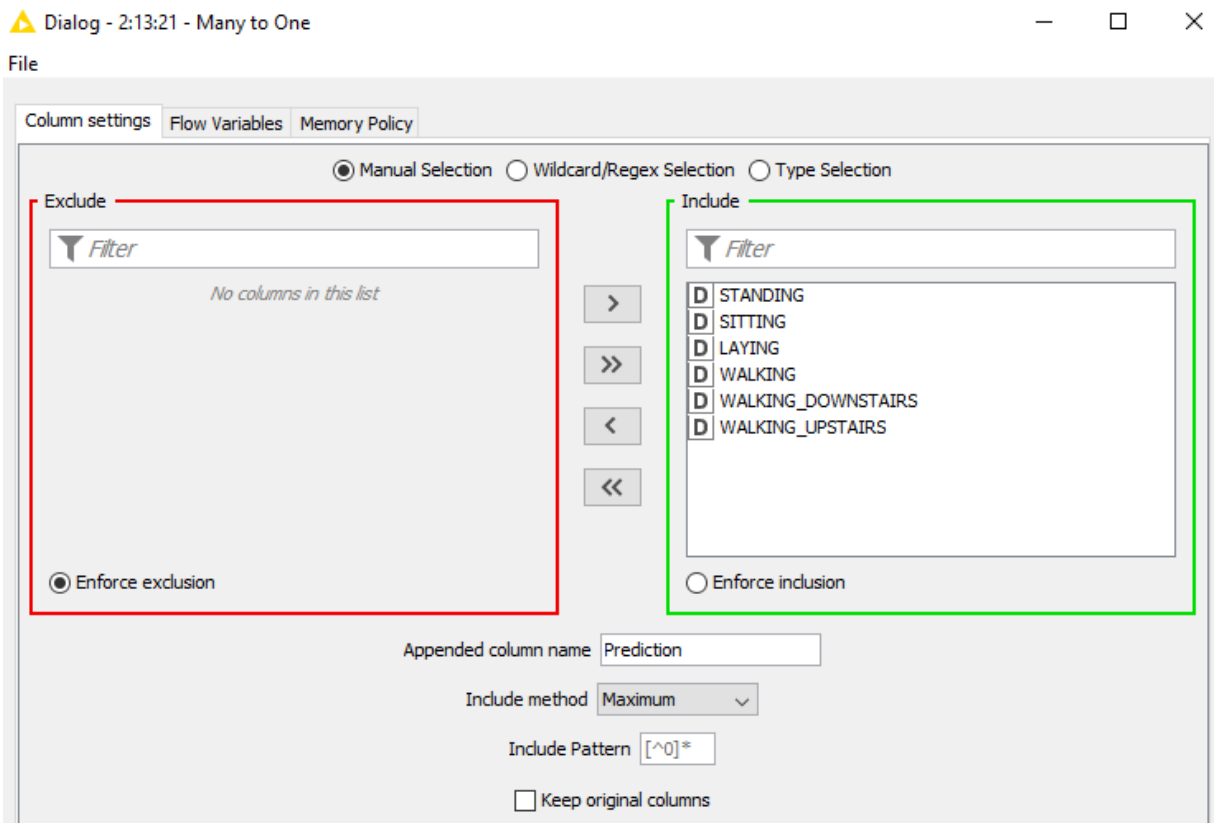
We see that we do not get one single prediction, but rather a probability score over the 6 different output possibilities. So we have to do the following:



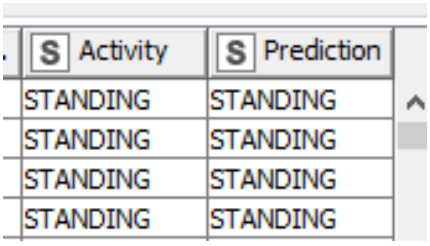
filter out these 6 output variables



renaming the outputs back into “STANDING, SITTING, LAYING, WALKING, WALKING_DOWNSTAIRS,WALKING_UPSTAIRS”



Then choose the maximal value out of the 6, renaming the resulting column “Prediction” and getting rid of the original columns.

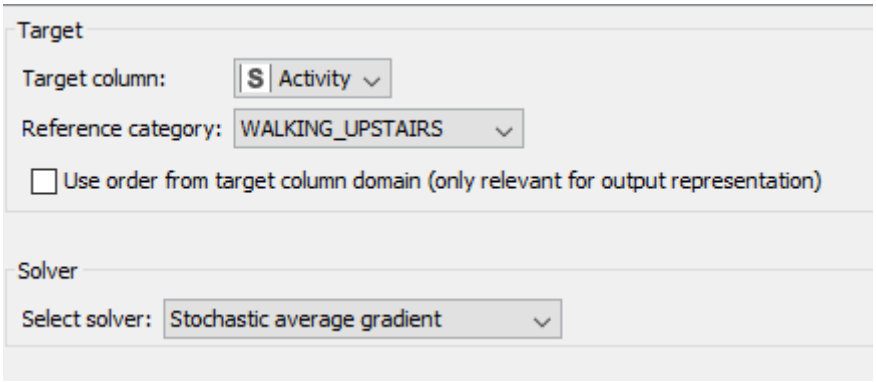


Activity	Prediction
STANDING	STANDING
STANDING	STANDING
STANDING	STANDING
STANDING	STANDING

In the final table, we have the Activity Column next to the Prediction column, and therefore a format that can be used for model evaluation!

7.3.2 The Logistic Regression

Logistic Regression is a special case of Linear Regression for Classification tasks, where we are using log of odds as dependent variable. It finds the probability of an event by fitting a logit function.



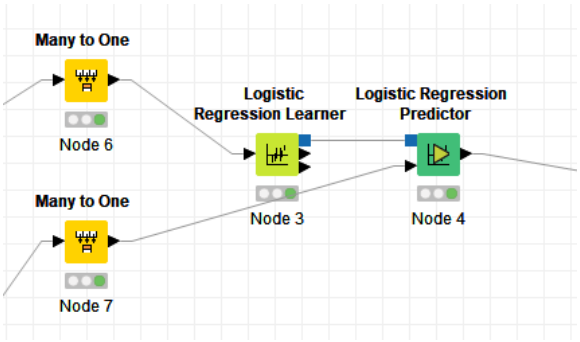
In order to understand what the solver option in a Logistic Regression means, we have to first think of what we try to accomplish with the logistic Regression.

$$h_\theta(x) = \frac{1}{1 + e^{\theta^\top x}}$$

This is the function that calculates the probabilities for being in which class. We want to choose the parameter (in this case omega) in a way that we make the least classification errors.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$
$$= -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]$$

So we end up with an Objective Function (or Cost Function) like that which we want to minimize. The solver basically is the way we choose to minimize this function, therefore we choose Stochastic Gradient Descent.



We end up with these nodes inside the Logistic Regression V1 Metanode (the V2 is the same just without undoing the one-hot encoding).

7.4 Model Evaluation

In order to compare multi-class classification algorithms, certain considerations have to be made.

In general, we would like to have a model that has a high Precision as well as Recall score. In reality, there is always a trade-off between the two.
Remember:
Precision = TP/TP+FP
Recall = TP / FN+TP

So, in order to find a metric that packs both measurements into one, the F1-score is used, which is described for a binary classification problem as follows:

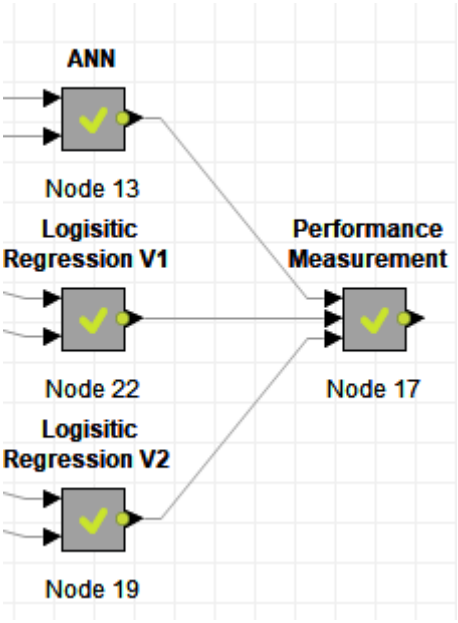
F1-score = 2 × (precision × recall)/(precision + recall)

In a multi-class classification, we can calculate the F1 score for every single class. So in our example, we can calculate the F1-score for the class STANDING, then for the class SITTING and so on.

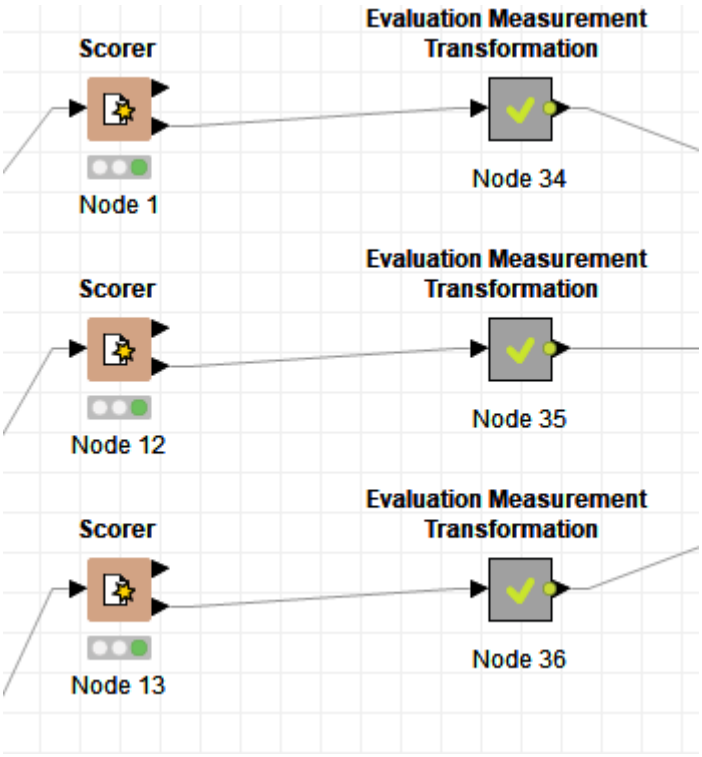
We then can combine these F1-scores into one **Macro-F1-score** by taking the arithmetic mean.

Another variante is to look at all the samples together and calculate one Precision and one Recall, which will result in the **Micro-F1-score** (which is the same as the model's accuracy).

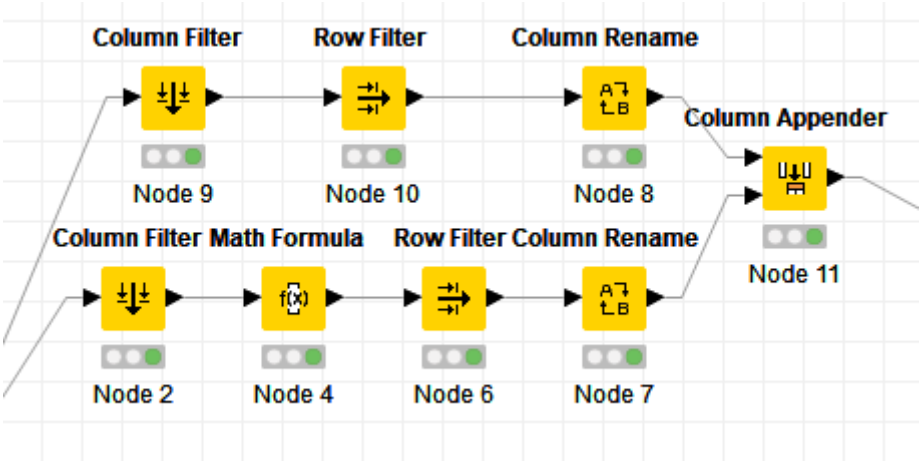
We normally use the Micro score when the classes are unbalanced. This means in our case we should get pretty similar results for both F1-score variants.



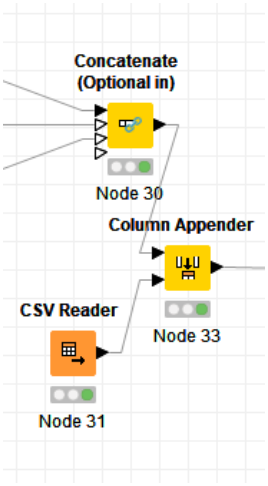
We can see here that we use another metanode in order to calculate comparable metrics for all three variants of models.



Looking inside the “Evalutation Measurement Transformation”:



We use the Scorer node to get the Micro-F1-Score (Accuracy score) on the one hand, and use the F-scores per class to calculate the Macro-F1-score. Ultimately we create a table containing only the two values.



We then use the three tables, combine them, and add the Row Tags (ANN, Logistic Regression V1 and V2) to the final table. The Final Output is as follows:

Row ID	D Micro F-Measure	D Macro F-Measure
ANN	0.938	0.937
Log RegV1	0.934	0.933
Log Reg V2	0.891	0.889

7.5 Implications for Data Scientists and Mangers

7.5.1 Interpretation of the Results and Implications for the Data Scientist

We can actually see that we have nearly no difference between the ANN and the Logistic Regression which used the same amount of Principal Components as the ANN, and also very good F1-scores for the Logistic Regression Model that only uses 26 attributes to make predictions.

This proves that sometimes, classic Machine Learning Algorithms are a good enough fit for the dataset at hand, and a Neural Network can not increase performance.
In addition to that, there are certain disadvantages to use ANNs where classic methods can do the job:

- Neural Networks are very hard to optimize to the dataset. Either it takes an extremely long time and computational effort (Grid Search) or one is required to code a complicated alternative (Bayesian Optimizing)
- Once they are optimized, they take longer to train than classical models, since they have more complexity within
- They often have a proneness to overfitting, which can be conquered with Dropout Layers, but that increases the complexity and the computational effort of the model
- They are overall more complex to design than classical methods

So, this experiment should show that choosing a neural network is only a great idea if we cannot achieve already good results with classical methods. A quick cross-validation of different algorithms with just using their default configuration normally already points into the right direction. Only if you cannot achieve high accuracy in that step, you should try to get into Deep Learning.

7.6 Implications for Managers

Generally speaking, from a business point of view the results achieved on this dataset are exceptionally good: we are able to predict the activity of our human respondents very well, by just using sensor data of a smartphone.

If the domain of activities measured can be extended (including more activites like “standing still”, “arm movement”, etc.), such human activity recognition analysis could be used by managers who are in charge of a high amount of workers, to monitor whether or not they are actively doing what they should be doing right now. One can envision the company to demand wearing a simple “smartwatch” on the wrist, with nothing more than accelerometers and gyroscopes, in order to track and measure the performance during work, incorporating these monitoring statistics into decision on promotions and job-cut-backs.

For managers in the health care industry, especially health care insurance companies, Human Activity Recognition can be a promising model to determine and adjust premium payments. Rewarding exercise activities while penalizing activities which are unhealthy (such as smoking etc.) could generate a fair evaluation of premium payments and help overcome the problem of moral hazard in the insurance industry (where customers who have insurance may be more likely to behave recklessly than those who do not have insurance, therefore driving premiums up for all insured people).

8 Works Cited

Corson, Nathalie. Asymptotic dynamics for slow-fast Hindmarsh-Rose neuronal system. 2009.

Cybenko, G. "Approximations by superpositions of sigmoidal functions." Mathematics of Control, Signals and Systems (1989): 303-314.

Gartner. Gartner.com. n.d. <https://www.gartner.com/en/research/methodologies/gartner-hype-cycle>.

Hebb, Donald O. The Organization of Behavior. New York: Wiley & Sons, 1949.

McCulloch, Warren and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity." Bulletin of Mathematical Biophysics (1943): 115-133.

Oezdemir, Deniz. On the Meaning of Passing Turing Tests through Generative Adversarial Networks under Human-Cognition Assumptions . 2020.

Pulvermüller, Friedemann. "Thinking in circuits: toward neurobiological explanation in cognitive neuroscience." Biological Cybernetics (2014).