It’s March 15th and that means it’s World Sleep Day (WSD). Don’t snooze off just yet! We’re about to check out a package that can make using R and Python a dream. It’s called reticulate and we’ll use it to train a Support Vector Machine for a simple classification task.

**What’s WSD got do with data science?**

I discovered WSD on the train to [STATWORX](https://www.statworx.com/) HQ in Frankfurt. If you’ve travelled by train before, you know that the only thing worse than disgruntled staff is unconscious staff. It hasn’t happened to me yet, but hey let’s face it: the personnel shortage of Deutsche Bahn isn’t getting better any time soon.

This brings me to the topic of today’s blog post: the sleeping habits of railroad workers. For this, I pulled a dataset from the US Federal Railroad Administration (FRA). The FRA conducted a survey on the work and sleep  
schedules of its train dispatchers. You can find it [here](https://www.fra.dot.gov/eLib/Details/L04105).

As the title suggests, we’re going to use both R and Python to predict whether a dispatcher was diagnosed with a sleeping disorder. Before we get started, a  
warning to all R and Python purists out there: the example below is a bit contrived. Everything we’re about to do can be done entirely in either one of the languages.

**So why bother using R *and* Python?**

As data scientists, we ideally have a firm grasp on both R and Python. After all, as my colleague Fran will discuss in an upcoming post, both have unique strengths that we can use to our advantage. Personally, I love R for the tidyverse and ggplot2, Python for its unified machine learning (ML) API scikit-learn.

Alright, point taken you might say, but it’s a hassle to switch back and forth. It’s not worth it, and until a few years ago you would have been right! However, nowadays there are some cool ways to get the best of both worlds. If you’re coming from the R community look no further than reticulate!

The reticulate package gives you a set of tools to use both R and Python interactively within an R session. Say you’re working in Python and need a specialized statistical model from an R package – or you’re working in R and want to access Python’s ML capabilities. Well, you’ve come to the right place. This package is a godsend for tasks where it’s required, or simply more convenient, to use both languages as part of your workflow.

Now, there are different ways to use R and Python interactively and I encourage you to check reticulate’s [github](https://github.com/rstudio/reticulate) site to see which one suits you best. In this post, we’re going through a simple example of how to use Python modules within an R Notebook (i.e. Markdown document).

**How to get started?**

If you’re working on a Mac, Python comes preinstalled on your system. Irrespective of the machine you’re on though, I recommended downloading the Anaconda distribution, so you have everything you need. One more note: you need RStudio’s newest preview version 1.2 for this to work. That’s it!

Let’s open an R Notebook, insert an R chunk and (install and) load the reticulate library. Immediately after loading reticulate, use the use\_python() command with the appropriate path. Placing it later in the script causes problems for some people. I specified my path as follows (note: yours may differ):

library(tidyverse)

library(recipes)

library(reticulate)

use\_python("/anaconda3/bin/python")

We’re good to go. Let’s read in the data and perform some transformations with dplyr. This is mostly recoding work. As you can see in the select command, we pick a handful of variables like sex, age, caffeine consumption, health and stress to predict whether a railroad dispatcher was diagnosed with a sleeping disorder.

**1) Read in the data**

data <- readxl::read\_xls("sleep.xls")

sleep <- data %>%

select(

Diagnosed\_Sleep\_disorder, Age\_Group, Sex, Total\_years\_dispatcher,

Total\_years\_present\_job, Marital\_Status, Childrendependents,

Children\_under\_2\_yrs, Caff\_Beverages, Sick\_Days\_in\_last\_year,

Health\_status, Avg\_Work\_Hrs\_Week, FRA\_report, Phys\_Drained,

Mentally\_Drained, Alert\_at\_Work, Job\_Security

) %>%

rename\_all(tolower) %>%

mutate\_if(is.character, as.numeric) %>%

mutate\_at(vars(diagnosed\_sleep\_disorder, sex, caff\_beverages, fra\_report),

~ -(. - 2)) %>%

mutate\_at(vars(marital\_status), ~ (. - 1)) %>%

drop\_na()

Now that we have the variables we want, it’s time to get the data into the right shape. Here’s one more reason to love R: the recipes package. If you’re not familiar with it, check it out. You may find its workflow a bit peculiar at first, but once you get used to it, it makes data preparation a breeze.

What we’re doing here is straightforward. First, we split the data into a training and test set. Next, we specify a data preparation recipe, which consists of three steps: one hot encoding factor predictors, standardising numeric predictors and down-sampling the data. One hot encoding and standardising ensure that the Support Vector Machine algorithm works properly. Down-sampling is a counter-measure against the class imbalance in our dataset.

**2) Prepare the data**

numeric\_variables <- c(

"total\_years\_dispatcher", "total\_years\_present\_job",

"childrendependents", "children\_under\_2\_yrs",

"sick\_days\_in\_last\_year", "avg\_work\_hrs\_week"

)

factor\_variables <- setdiff(colnames(sleep), numeric\_variables)

sleep <- mutate\_at(sleep, vars(factor\_variables), as.factor)

set.seed(2019)

index <- sample(1:nrow(sleep), floor(nrow(sleep) \* .75))

sleep\_train <- sleep[index, ]

sleep\_test <- sleep[-index, ]

recipe\_formula <- recipes::recipe(diagnosed\_sleep\_disorder ~ ., sleep\_train)

recipe\_steps <- recipe\_formula %>%

recipes::step\_dummy(factor\_variables, -all\_outcomes(), one\_hot = TRUE) %>%

recipes::step\_downsample(diagnosed\_sleep\_disorder) %>%

recipes::step\_center(numeric\_variables) %>%

recipes::step\_scale(numeric\_variables)

recipe\_prep <- recipes::prep(recipe\_steps, sleep\_train, retain = TRUE)

training\_data <- juice(recipe\_prep)

testing\_data <- bake(recipe\_prep, sleep\_test)

Now comes the part where Python shines: its unified ML library scikit-learn. Let’s go ahead and import the Support Vector Machine (SVM) classifier as well as some other modules to tune and evaluate our model.

SVM is a supervised learning algorithm. It works by finding a hyperplane in an N-dimensional space, which separates two (or more) classes as cleanly as possible. More technically, SVM maximizes the margin or the distance between the separating hyperplane and the closest data points. This is why SVM is also called a maximum margin estimator.

SVM is mostly used for classification, but it can do regression too. The upside is that it works with high-dimensional data and different kernel functions, meaning it can flexibly adapt to different types of data. Its downside is that computation becomes costly with large datasets and that it reacts sensitively to hyperparameters. Still, for some applications SVM performs quite competitively.

Combining SVM with kernels allows us to project our data into a higher-dimensional space. The point of this is to make the classes better separable. In our example here, we’ll use a simple linear and a radial basis function kernel. The latter can map the predictor space into infinite dimensions.

**3) Train the model**

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn import svm

from sklearn.model\_selection import GridSearchCV, cross\_val\_score

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

y\_train = r.training\_data['diagnosed\_sleep\_disorder']

X\_train = r.training\_data.drop('diagnosed\_sleep\_disorder', axis = 1)

y\_test = r.testing\_data['diagnosed\_sleep\_disorder']

X\_test = r.testing\_data.drop('diagnosed\_sleep\_disorder', axis = 1)

clf = svm.SVC(kernel = 'linear')

clf.fit(X\_train, y\_train)

y\_pred = clf.predict(X\_test)

print(confusion\_matrix(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred))

clf = svm.SVC(kernel = 'RBF')

clf.fit(X\_train, y\_train)

y\_pred = clf.predict(X\_test)

print(confusion\_matrix(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred))

We can tune an SVM with the help of two parameters: C and gamma. C is a regularisation parameter that represents how much error we’re willing to tolerate. The higher C, the stricter we are, the more exactly SVM will try to fit the data by picking a hyperplane with smaller margins. As a consequence, higher C’s carry a higher risk of overfitting. For the radial basis function kernel, we can also tune gamma which determines the size of the kernel and with it the decision boundary. The higher gamma, the closer the decision boundary is to single training examples. A higher gamma can lead to a better model, yet likewise increases the risk of overfitting.

**4) Tune the model**

param\_grid = [{'C': [0.01, 0.1, 1, 10, 100],

'gamma': [0.001, 0.01, 0.1, 1, 10],

'kernel': ['RBF', 'linear']}]

grid = GridSearchCV(svm.SVC(), param\_grid, cv = 5, scoring = 'balanced\_accuracy')

grid.fit(X\_train, y\_train)

print(grid.best\_params\_)

From the cross-validation procedure, it appears that a gamma of 0.1 and a C of 0.01 are optimal in combination with a radial basis function kernel. So, let’s train our model with this kernel and the hyperparameter values.

**5) Evaluate the accuracy of the model**

clf = grid.best\_estimator\_

y\_pred = clf.predict(X\_test)

print('Confusion Matrix:\n\n', confusion\_matrix(y\_test, y\_pred))

print('\nClassification Report:\n\n', classification\_report(y\_test, y\_pred))

print('\nTraining Set Accuracy: {:.2f}%'.format(clf.score(X\_train, y\_train)))

print('\nTest Set Accuracy: {:.2f}%'.format(clf.score(X\_test, y\_test)))

conf\_mat = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(conf\_mat, square = True, annot = True, fmt = 'g',

cbar = False, cmap = 'viridis')

plt.xlabel('predicted')

plt.ylabel('observed')

plt.show()

In this case, we achieve a training set accuracy of 93 per cent and a test set accuracy of 74 per cent. This suggests that some overfitting has occurred. To achieve a higher accuracy (or better: sensitivity/recall), we could experiment with different kernels and/or hyperparameter values. But this I’d leave up to you. With reticulate, you now have a tool to get the best of both R and Python.