Lab 3 - Machine Learning for Social Science (Solutions)

To be handed in no later than September 17th 10:00. The submission should include code, relevant output, as well as answers to questions. We recommend the use of RMarkdown to create the report.

Part 1: Salary prediction

In the first part of this lab, we will work with a dataset (`adults`) containing information about a sample of US individuals' salaries as well as a handful sociodemographic variables. We will also consider an augmented version of this dataset (`adults_aug`), combining the original data with (simulated) highly nuanced lifecourse information. With this, the objective is to explore how accurately we can predict the salaries of individuals, comparing neural networks with other methods from previous weeks.

1. Begin by importing `adults.rds` and partition the data into a *training* and *test* set. We will use the training set to fit our model, and the test set to assess accuracy and compare across models. The dataset contains cirka 50,000 individuals, and for each we have information about five traditional variables (age, education, hours worked per week, capital gain and capital loss). We are reasonably sure there are no complex interactions or non-linearities. With this as the background, do you believe a neural network model is likely to excel on this dataset? Why/why not?

Based on the provided information — that the data contain few variables, and that domain knowledge suggests their relationship with the outcome is not overly complex — suggest that we may get rather small gains from using (deep) neural networks compared to a standard linear model. The fact that we do have a substantial number of observations relative to the number of variables, though, should allow the model to robustly identify any smaller non-linearities that may exist, which may provide a small improvement.

```
# Import
adults <- readRDS(file = 'data/final/adults.rds')
# Train / test split (80/20)
adults_X <- adults[,c(1:5)]
adults_y <- adults[,6]
n <- nrow(adults_X)
set.seed(1234)
idx <- sample(seq_len(n), size = floor(0.8 * n))
X_train <- adults_X[idx, , drop = FALSE]
y_train <- adults_y[idx]
X_test  <- adults_X[-idx, , drop = FALSE]
y_test  <- adults_y[-idx]
input_dim <- ncol(X_train)
```

2. Begin with estimating a standard logistic regression model to the training set you just created. Hint: you may use the glm(..., family='binomial') function to estimate a standard logistic regression model. When estimation has finished, use the predict() function to predict the outcome on the test dataset, and calculate (report) the accuracy.

```r
# Standard logit model
logit <- glm(formula = y ~ .,
             data = data.table(X_train,y=y_train),
             family = 'binomial')
logit_preds <- predict(object = logit,
                       newdata = as.data.table(X_test))
logit_preds <- ifelse(logit_preds>0.5, 1, 0)
logit_accuracy <- mean(logit_preds==y_test)
print(logit_accuracy)
```

```
## [1] 0.8043812
```

3. Estimate a neural network with 1 *hidden layer* and 5 *hidden units*. In the `compile()` function, set the optimizer to `optimizer_adam()`, the loss function equal to `"binary_crossentropy"` (as the outcome is binary), and metrics to `"accuracy"`. Report the accuracy. Did the result match your expectations formulated in #1?

Adding a hidden layer only provides a marginal improvement. This is in line with expectations.

```r
# NN with 1 hidden layer
k_clear_session()
set.seed(1234)
nn1 <- keras_model_sequential() |>
  layer_dense(units = 5, activation = "relu", input_shape = input_dim) |>
  layer_dense(units = 1, activation = "sigmoid")

nn1 |>
    compile(
      optimizer = optimizer_adam(),
      loss      = "binary_crossentropy",
      metrics   = "accuracy")

history_nn1 <- nn1 |>
  fit(
    x = X_train, y = y_train,
    epochs = 50,
    batch_size = 128,
    validation_split = 0.2,
    verbose = 0
  )

# Evaluate on holdout
nn1_preds <- as.numeric(nn1 |> predict(X_test))
```

```
## 306/306 - 0s - 202us/step
```

```r
nn1_preds <- ifelse(nn1_preds>0.5, 1, 0)
nn1_accuracy <- mean(nn1_preds==y_test)
print(nn1_accuracy)
```

```
## [1] 0.8163579
```

4. Next, you shall estimate a considerably more complex neural network, containing 4 *hidden layers*. The first hidden layer should have 256 *hidden units*, the second hidden layer 128 hidden units, the third hidden layer 64 hidden units, and the fourth hidden layer 32 hidden units. Use the same settings for `compile()` as in #3. Report the *total number of parameters* and then estimate the model. Do you find that it outperforms #3 meaningfully? Why do you think this is (or is not) the case?

```
# NN with 4 hidden layer
k_clear_session()
set.seed(1234)
nn2 <- keras_model_sequential() |>
  layer_dense(units = 256, activation = "relu", input_shape = input_dim) |>
  layer_dense(units = 128, activation = "relu") |>
  layer_dense(units = 64, activation = "relu") |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dense(units = 1, activation = "sigmoid")

print(nn2)
```

```
## Model: "sequential"
##
##  Layer (type)                        Output Shape                    Param #
##
##  dense (Dense)                       (None, 256)                       1,536
##
##  dense_1 (Dense)                     (None, 128)                      32,896
##
##  dense_2 (Dense)                     (None, 64)                        8,256
##
##  dense_3 (Dense)                     (None, 32)                        2,080
##
##  dense_4 (Dense)                     (None, 1)                            33
##
##  Total params: 44,801 (175.00 KB)
##  Trainable params: 44,801 (175.00 KB)
##  Non-trainable params: 0 (0.00 B)
```

Specified this way, the neural network contains 44,801 parameters to be estimated.

```
nn2 |>
    compile(
      optimizer = optimizer_adam(),
      loss      = "binary_crossentropy",
      metrics   = "accuracy")

history_nn2 <- nn2 |>
  fit(
    x = X_train, y = y_train,
    epochs = 50,
    batch_size = 128,
    validation_split = 0.2,
    verbose = 0
  )
```

```r
# Evaluate on holdout
nn2_preds <- as.numeric(nn2 |> predict(X_test))
```

```
## 306/306 - 0s - 279us/step
```

```r
nn2_preds <- ifelse(nn2_preds>0.5, 1, 0)
nn2_accuracy <- mean(nn2_preds==y_test)
print(nn2_accuracy)
```

```
## [1] 0.8236258
```

We again see a marginal, but non-neglibile improvement in performance. A neural network with a single hidden layer and 5 hidden units remains a relatively simple model (36 parameters). Expanding it to a 44,801 parameter model is a substantial increase in flexibility, enabling the model to pick up on complex non-linearities and interaction effects. However, because the data does not contain overly complex patterns, the gain is marginal.

5. Suppose now that we retrieve a dataset (`adults_aug`) that expands upon the original 'adults dataset. This dataset contains 6 extra variables; which capture complex aspects of the individuals life courses, with various interdependencies between them and possible non-linearities. Could this expanded data make a difference in terms of more clearly outperforming the standard logit? Investigate this by repeating steps 2-4 on the adults_aug dataset. Does your conclusion about the relevance of more complex network structure differ between the two datasets? Why?

```r
# Import
adults_aug <- readRDS(file = 'data/final/adults_aug.rds')
# Train / test split (same idx)
adults_aug_X <- adults_aug[,c(1:11)]
adults_aug_y <- adults_aug[,12]
X_train <- adults_aug_X[idx, , drop = FALSE]
y_train <- adults_aug_y[idx]
X_test  <- adults_aug_X[-idx, , drop = FALSE]
y_test  <- adults_aug_y[-idx]
input_dim <- ncol(X_train)
```

```r
# 2: Standard logit model
logit <- glm(formula = y ~ .,
             data = data.table(X_train,y=y_train),
             family = 'binomial')
logit_preds <- predict(object = logit,
                       newdata = as.data.table(X_test))
logit_preds <- ifelse(logit_preds>0.5, 1, 0)
logit_accuracy <- mean(logit_preds==y_test)
print(logit_accuracy)
```

```
## [1] 0.8046883
```

```r
# 3: NN with 1 hidden layer
k_clear_session()
set.seed(1234)
```

```
nn1 <- keras_model_sequential() |>
  layer_dense(units = 5, activation = "relu", input_shape = input_dim) |>
  layer_dense(units = 1, activation = "sigmoid")

nn1 |>
    compile(
      optimizer = optimizer_adam(),
      loss      = "binary_crossentropy",
      metrics   = "accuracy")

history_nn1 <- nn1 |>
  fit(
    x = X_train, y = y_train,
    epochs = 50,
    batch_size = 128,
    validation_split = 0.2,
    verbose = 0
  )

# Evaluate on holdout
nn1_preds <- as.numeric(nn1 |> predict(X_test))
```

```
## 306/306 - 0s - 200us/step
```

```
nn1_preds <- ifelse(nn1_preds>0.5, 1, 0)
nn1_accuracy <- mean(nn1_preds==y_test)
print(nn1_accuracy)
```

```
## [1] 0.920258
```

```
# 4: NN with 4 hidden layer
k_clear_session()
set.seed(1234)
nn2 <- keras_model_sequential() |>
  layer_dense(units = 256, activation = "relu", input_shape = input_dim) |>
  layer_dense(units = 128, activation = "relu") |>
  layer_dense(units = 64, activation = "relu") |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dense(units = 1, activation = "sigmoid")

nn2 |>
    compile(
      optimizer = optimizer_adam(),
      loss      = "binary_crossentropy",
      metrics   = "accuracy")

history_nn2 <- nn2 |>
  fit(
    x = X_train, y = y_train,
    epochs = 50,
    batch_size = 128,
    validation_split = 0.2,
```

```
    verbose = 0
  )

# Evaluate on holdout
nn2_preds <- as.numeric(nn2 |> predict(X_test))
```

```
## 306/306 - 0s - 280us/step
```

```
nn2_preds <- ifelse(nn2_preds>0.5, 1, 0)
nn2_accuracy <- mean(nn2_preds==y_test)
print(nn2_accuracy)
```

```
## [1] 0.9951889
```

Here we see substantial improvements — at each step. The addition of a single hidden layer provides a significant boost; and adding even more complexity (36 → 45K parameters) further improves the results. With the most complex model, we are almost perfectly predicting the individuals' salaries (whether they are above/below the median). If this was a real dataset, this could for example indicate that there are intricate path dependencies in the life-course, between life-events and life-outcomes, that are not well captured by a linear model, which seems plausible. So, addressing the question: the utility of a more complex model is indeed different here compared to before. The reason is that the augmented dataset contains within it complex interactions and non-linearities.

6. Given the results, you may find it unnecessary to to try further revisions. But for learning purposes, suppose we want to re-estimate the best-performing model on the adults_aug data using *dropout learning*. Update the model to perform dropout for the first three hidden layers. Estimate one model where you inactivate 10% for each of the hidden layers, and a second model where you inactivate 90% per layer, and report the results. In what direction does the results change? Do you attribute this change to a change in bias or in variance and why?

```
# Dropout 10%
k_clear_session()
set.seed(1234)
nn3 <- keras_model_sequential() |>
  layer_dense(units = 256, activation = "relu", input_shape = input_dim) |>
  layer_dropout(rate = 0.1) |>
  layer_dense(units = 128, activation = "relu") |>
  layer_dropout(rate = 0.1) |>
  layer_dense(units = 64, activation = "relu") |>
  layer_dropout(rate = 0.1) |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dropout(rate = 0.1) |>
  layer_dense(units = 1, activation = "sigmoid")

nn3 |>
    compile(
      optimizer = optimizer_adam(),
      loss      = "binary_crossentropy",
      metrics   = "accuracy")

history_nn3 <- nn3 |>
```

```
  fit(
    x = X_train, y = y_train,
    epochs = 50,
    batch_size = 128,
    validation_split = 0.2,
    verbose = 0
  )

# Evaluate on holdout
nn3_preds <- as.numeric(nn3 |> predict(X_test))
```

```
## 306/306 - 0s - 279us/step
```

```
nn3_preds <- ifelse(nn3_preds>0.5, 1, 0)
nn3_accuracy <- mean(nn3_preds==y_test)
print(nn3_accuracy)
```

```
## [1] 0.9973385
```

```
# Dropout 90%
k_clear_session()
set.seed(1234)
nn4 <- keras_model_sequential() |>
  layer_dense(units = 256, activation = "relu", input_shape = input_dim) |>
  layer_dropout(rate = 0.90) |>
  layer_dense(units = 128, activation = "relu") |>
  layer_dropout(rate = 0.90) |>
  layer_dense(units = 64, activation = "relu") |>
  layer_dropout(rate = 0.90) |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dropout(rate = 0.90) |>
  layer_dense(units = 1, activation = "sigmoid")

nn4 |>
    compile(
      optimizer = optimizer_adam(),
      loss      = "binary_crossentropy",
      metrics   = "accuracy")

history_nn4 <- nn4 |>
  fit(
    x = X_train, y = y_train,
    epochs = 50,
    batch_size = 128,
    validation_split = 0.2,
    verbose = 0
  )

# Evaluate on holdout
nn4_preds <- as.numeric(nn4 |> predict(X_test))
```

```
## 306/306 - 0s - 282us/step
```

```
nn4_preds <- ifelse(nn4_preds>0.5, 1, 0)
nn4_accuracy <- mean(nn4_preds==y_test)
print(nn4_accuracy)
```

```
## [1] 0.9522981
```

For the 10% dropout model, we don't see much of a difference. The already excellent hold-out performance of the original model suggests that it cannot be meaningfully overfitted, and as such, dropout have limited scope to provide further improvements.

For the 90% dropout model, we see a significant ($\approx$ 8 pct point) drop in performance. This can be attributed to an increase in bias: by inactivating as much as 90% of the weights at each itertion of the estimation, we are regularizing the model rather severly, forcing it to learn more generic/robust weights, and degrading its capacity to learn complex patterns.

Still, I'm a bit surprised that it does as well as it is. My guess would be that this is because the total number of parameters is so large, that 10% still corresond to 5,000 or so non-zero weights (for a relatively small number of input variables). Below, I increase the dropout percentage even more (to 95%) and now we do see a considerable drop in performance.

```
# Dropout 95%
k_clear_session()
set.seed(1234)
nn5 <- keras_model_sequential() |>
  layer_dense(units = 256, activation = "relu", input_shape = input_dim) |>
  layer_dropout(rate = 0.95) |>
  layer_dense(units = 128, activation = "relu") |>
  layer_dropout(rate = 0.95) |>
  layer_dense(units = 64, activation = "relu") |>
  layer_dropout(rate = 0.95) |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dropout(rate = 0.95) |>
  layer_dense(units = 1, activation = "sigmoid")

nn5 |>
    compile(
      optimizer = optimizer_adam(),
      loss      = "binary_crossentropy",
      metrics   = "accuracy")

history_nn5 <- nn5 |>
  fit(
    x = X_train, y = y_train,
    epochs = 50,
    batch_size = 128,
    validation_split = 0.2,
    verbose = 0
  )

# Evaluate on holdout
nn5_preds <- as.numeric(nn5 |> predict(X_test))
```

```
## 306/306 - 0s - 289us/step
```

```r
nn5_preds <- ifelse(nn5_preds>0.5, 1, 0)
nn5_accuracy <- mean(nn5_preds==y_test)
print(nn5_accuracy)
```

```
## [1] 0.7516634
```

Part 2: image prediction

In the second part of the lab, we'll work with *Fashion MNIST data* (https://www.kaggle.com/datasets/zalando-research/fashionmnist) containing images of 10 different types of fashion items — including t-shirts, coats, bags, and sneakers. All images are 28 pixels in width and 28 pixels in height (784 in total). Each pixel has a single value associated with it; its gray-scale value ranging between 0 and 255.

For the purpose of this lab, we are imagining a fictive scenario in which each image correspond to a consumed good, and that we have (simulated) *basic socioeconomic information* about the *customer* who bought the fahion item, as well as the *year* it was consumed. Our social scientific interest is to examine whether there are any trends in the association between types of fashion goods and the socioeconomic status of the customer. For example, are sneakers more or less associated with low/high economic status, and has this changed over time?

The practical problem that we face, is that we only have labeled data of the images (i.e., identifying which fashion item is in the image) for part of the time-period (in 2016 but not in 2017). Our objective, thus, is to use *deep learning* to estimate a model on the part of the data where we do have labeled image data, and then use the trained model to predict on the yet to be labeled images.

1. Begin by importing the datafiles *"fashion_2016_train.rds"* and *"fashion_2016_test.rds"*.

```r
# Import
train_2016 <- readRDS(paste0(mywd,"data/fashion_2016_train.rds"))
test_2016 <- readRDS(paste0(mywd,"data/fashion_2016_test.rds"))
x_train <- train_2016$images
y_train <- train_2016$labels
x_test <- test_2016$images
y_test <- test_2016$labels

# Normalize pixel values
x_train <- x_train / 255
x_test <- x_test / 255

# Add channel dimension (Fashion MNIST is grayscale)
x_train <- array_reshape(x_train, c(dim(x_train), 1))
x_test <- array_reshape(x_test, c(dim(x_test), 1))

# Dims
n_train <- dim(x_train)[1]
n_test <- dim(x_test)[1]
```

2. Next, estimate a simple convolutional neural network with K=1 convolutional layers and M=1 regular type of (fully connected) hidden layers. Specify the *number of filters* (in the convolutional layer) and the *number of hidden units* (in the fully connected / regular hidden slides) to be 8. Remember also that included after each added convolutional layer, *pooling* (max-pooling, $2 * 2$) should to be applied. Finally, include `layer_dense(units = 10, activation = "softmax")` at the end. When

compiling the model using the compile() function, set loss = "sparse_categorical_crossentroy". Report the results, and reflect on whether you think improvement can be made by increasing either M, K, or the number of filters or hidden units in the regular hidden layer.

```
# Simple CNN (M=1; K=1); 8 hidden units in dense layer, 8 filters.
k_clear_session()
set.seed(1234)
cnn1 <- keras_model_sequential() %>%
  # First convolutional block
  layer_conv_2d(filters = 8,
                kernel_size = c(3, 3),
                activation = "relu",
                input_shape = c(28, 28, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # Dense layers
  layer_flatten() %>%
  layer_dense(units = 8, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")  # 10 fashion categories

# Compile model
cnn1 %>% compile(
  optimizer = optimizer_adam(),
  loss = "sparse_categorical_crossentropy",
  metrics = c("accuracy")
)

# Fit
history_cnn1 <- cnn1 |>
  fit(x_train,
      y_train,
      validation_split=0.2,
      epochs=50,
      batch_size=128,
      verbose=0)

# Accuracy
cnn1_test_accuracy <- cnn1 %>% evaluate(x_test, y_test, verbose = 0)
print(cnn1_test_accuracy)
```

```
## $accuracy
## [1] 0.8862554
##
## $loss
## [1] 0.3351633
```

Increasing the number of convolutional layers (M) and the number of filters enable the learning of more intricate patters in the images. Increasing the number of regular, fully-connected layers (K) enable learning more complex associations between the detected features of the images and the outcome classes. Our image data is relatively simple (it is easy to think of much more complex images, with more details in them), such that the potential for gains by increasing M or the number of filters is unclear; but worth evaluating. A similar case can be made for the number of regular, fully-connected hidden layers: do we suspect that there are complex associations between the learned image patterns and the object classes? Not so easy to say without knowing the image-features beforehand, of course. But, the complexity of the images and how *easily distinguishable* we think the different outcome classes are can inform this guess.

3. Next, you shall estimate a slightly more complex convolutional neural network, increasing the number of filters to 32 in the first layer, and 64 in the second convolutional layer. Similarly, increase the number of hidden units in the first regular/fully-connected hidden layer to 64, and the second to 32. Report the results. Does this slightly more complicated model improve/degrade upon the simple one in #2? Speculate why in terms of the bias/variance trade-off. Report and contrast also the number of parameters between the two.

```
# More sophisticated CNN (M=2; K=2)
k_clear_session()
set.seed(1234)
cnn2 <- keras_model_sequential() %>%
  # Convolutional layers
  layer_conv_2d(filters = 32,
                kernel_size = c(3, 3),
                activation = "relu",
                input_shape = c(28, 28, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64,
                kernel_size = c(3, 3),
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # Dense layers
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")  # 10 fashion categories

# Compile model
cnn2 %>% compile(
  optimizer = optimizer_adam(),
  loss = "sparse_categorical_crossentropy",
  metrics = c("accuracy")
)

# Fit
history_cnn2 <- cnn2 |>
  fit(x_train,
      y_train,
      validation_split=0.2,
      epochs=50,
      batch_size=128,
      verbose=0)

# Accuracy
cnn2_test_accuracy <- cnn2 %>% evaluate(x_test, y_test, verbose = 0)
print(cnn2_test_accuracy)
```

```
## $accuracy
## [1] 0.8987013
##
## $loss
## [1] 0.6172688
```

We see a slight improvement. To the extent that this is a robust difference, it should reflect a decrease in

bias — since this is a more complex model compared to the previous one. If we wanted to understand the source of the improvement in more detail, we could estimate additional models where we only add more layers and units for the convolutional layer, keeping the dense hidden layer simple, and vice versa. Note: had this been a real research setting, I would have performed a systematic grid search of the many parameter combinations.

4. In #2-3, we used max-pooling of 2*2. Why is it generally not a good idea to increase the dimension of the pooling to become large relative to the images? Would this increase bias or variance?

When we apply (max) pooling, we keep one (the largest) cell-value within the pooling window. If we use a large pooling window, we risk blurring out the distinctive patterns detected by the filter. In the extreme, if we have a pool window as large as the image itself then each feature map would just contain a single value, completely washing out any pattern detected. In terms of bias/variance, pooling across increasingly larger surfaces smoothens patterns out, and should thus increase bias.

5. Because we are interested in using the predictions from these models for downstream analysis (of trends in consumer polarization), it is extra important to validate the measure. This is what you shall do now, focusing on the CNN model you deemed the best thus far. In particular you should break down the accuracy per item category. Is there meaningful difference in predictability beween item classes? Does the ordering make substantive sense? Hint: to compute accuracy by group, you may use the following code:

```
predictions <- cnn2 %>% predict(x_test)
```

```
## 289/289 - 0s - 1ms/step
```

```
predicted_classes <- apply(predictions, 1, which.max) - 1
accuracy_by_class_dt <- data.table(actual=y_test,predicted=predicted_classes)
accuracy_by_class_dt[,correct := fifelse(actual==predicted, yes = 1, no = 0)]
fashion_labels <- c("T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
                    "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot")
fashion_labels_dt <- data.table(label=fashion_labels,number=0:9)
accuracy_by_class_dt <- merge(x=accuracy_by_class_dt,y=fashion_labels_dt,by.x='actual',by.y='number')
accuracy_by_class_dt[,.(accuracy=mean(correct)),by=label][order(accuracy,decreasing = T)]
```

```
##           label  accuracy
##          <char>     <num>
##  1:     Trouser 0.9789590
##  2:         Bag 0.9759162
##  3:      Sandal 0.9737130
##  4:  Ankle boot 0.9673558
##  5:     Sneaker 0.9435570
##  6:       Dress 0.9087003
##  7: T-shirt/top 0.8835616
##  8:    Pullover 0.8768197
##  9:        Coat 0.8677686
## 10:       Shirt 0.6161188
```

The classes the model struggles with are similar clothing items with relatively few distinctive features. By contrast, sandals, trousers, and bags have clearer cues and are easier to predict. In a research setting, I'd examine the 10x10 confusion matrix—relating true to predicted labels—to identify which categories are most often confused with which.

6. Next, we shall finally address the question we set out to answer: how patterns in consumer behavior changed between 2016 to 2017. To answer this question, please follow these steps:

- Import the `fashion2016_2017_unlabeled.rds` file, which contains all images and the sociodemographic info of the individual associated with the image. Note: the data has already been preprocced (normalized pixel values, etc.).
- Use the predict() function to predict for this dataset based on the CNN model you thought were the best. Note that, when you use the predict() function for a model where you have multiple-category outcome variables, like here, each observation get a probability over the items. To assign the prediction to the maximum value, you may use this code:

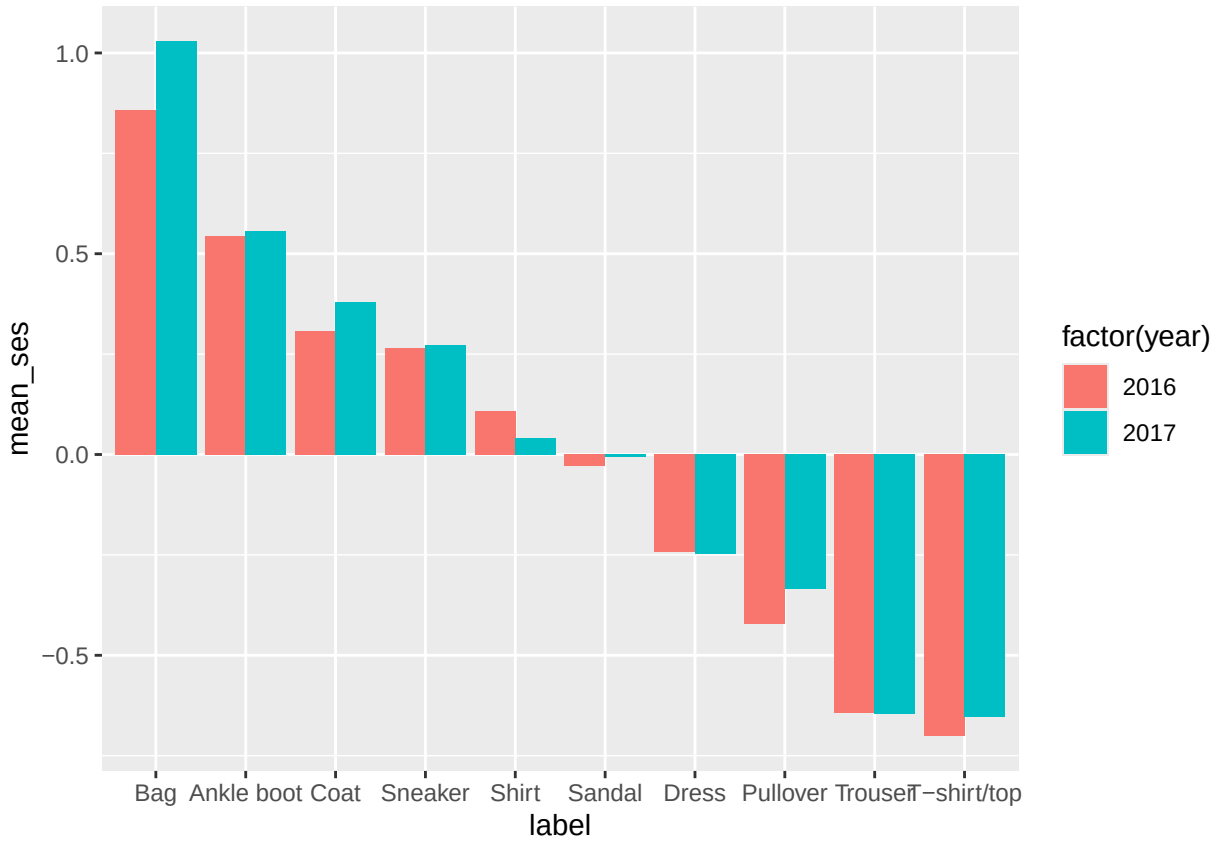
```
apply(preds_2016_2017, 1, which.max) - 1
```

- Then you can simply merge the predicted category with the demographic variable data.frame, and calculate various associations by classical statistical means.
- Report your findings.

```
# i) import
unlabeled_2016_2017 <- readRDS(paste0(mywd,"data/final/fashion_2015_2016_unlabeled.rds"))
x_unlabeled_2016_2017 <- unlabeled_2016_2017$images

# ii) predict
preds_2016_2017 <- cnn2 %>% predict(x_unlabeled_2016_2017)
```

```
## 2188/2188 - 2s - 1ms/step
```

```
pred_class_2016_2017 <- apply(preds_2016_2017, 1, which.max) - 1

# iii) merge ses and predicted item
consumption_ses_dt_1617 <- setDT(cbind(unlabeled_2016_2017$demographics,
                                pred_y=pred_class_2016_2017))
consumption_ses_dt_1617 <- merge(x=consumption_ses_dt_1617,
                                y=fashion_labels_dt,
                                by.x='pred_y',by.y='number')

# iv) calculate some basic associations
consumption_ses_dt_1617_2 <- consumption_ses_dt_1617[,.(mean_ses=mean(ses_score)),
                                                      by=.(label,year)]
# Basic plot
lab_order <- consumption_ses_dt_1617_2[
  , .(mx = max(mean_ses)), by = label
][order(-mx), label]
consumption_ses_dt_1617_2[, label := factor(label, levels = lab_order)]
ggplot(consumption_ses_dt_1617_2,aes(x=label,y=mean_ses,fill=factor(year))) +
  geom_bar(stat = 'identity', position='dodge')
```




```
# Gini
gini_zero <- function(x, na.rm = TRUE) {
  if (na.rm) x <- x[!is.na(x)]
  x <- as.numeric(x); n <- length(x)
  if (n < 2) return(0)
  m1 <- mean(abs(x))
  if (m1 == 0) return(0)
  sx <- sort(x)
  S  <- sum((2*seq_len(n) - n - 1) * sx)         # sum_{i<j} (x_j - x_i)
  gmd <- 2 * S / (n * (n - 1))                   # average |xi - xj|
  gmd / (2 * m1)
}
consumption_ses_dt_1617_2[,gini_zero(mean_ses),by=year]
```

```
##     year        V1
##    <num>     <num>
## 1:  2016 0.7459618
## 2:  2017 0.7664434
```

```
# Other
DT <- copy(consumption_ses_dt_1617)
setDT(DT)
DT[, `:=`(
  label  = factor(label),
  urban  = factor(urban, levels = c(0,1), labels = c("rural","urban")),
```

```
  region = factor(region),
  year   = factor(year)
)]

# Income density: compare years within each label
ggplot(DT, aes(x = income, fill = year, color = year)) +
  geom_density(alpha = 0.25, adjust = 1.1) +
  facet_wrap(~ label, scales = "free_y") +
  labs(title = "Income distributions by label, split by year",
       x = "Income", y = "Density", fill = "Year", color = "Year") +
  theme_minimal()
```