MLP Project 1 Submission

March 14, 2024

1 Machine Learning in Python - Project 1

Due Friday, March 8th by 4 pm.

Axel Eichelmann, Darragh Ferguson, Heather Napthine, Will Spence

1.1 Setup

[18]:
```
# Import all required libraries.
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
import feature_engine
import imblearn
import scipy
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import FunctionTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import OneHotEncoder
from sklearn import set_config
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import LassoCV
from feature_engine.transformation import LogTransformer
from imblearn.pipeline import Pipeline
from scipy import stats
from imblearn import FunctionSampler
```

```
from scipy.special import inv_boxcox
from numpy.linalg import solve

# Set plotting defaults.
plt.rcParams['figure.figsize'] = (8,5)
plt.rcParams['figure.dpi'] = 80
```

[19]:
```
# Load data in easyshare.csv and extra data from easyshare_all.csv
original_d = pd.read_csv("easyshare.csv")
extra_d = pd.read_csv("easyshare_all.csv")
```

2 Introduction

According to the World Health Organization, 55 million people currently suffer from dementia, with 10 million new cases annually. It is the seventh leading cause of death, and in 2019 alone, cost global economies 1.3 trillion US dollars. With no cure found, preventative measures against dementia are crucial in fighting this public health crisis. This report aims to identify key genetic and lifestyle factors for predicting dementia, predictions which allow for crucial insight into potential preventative measures for this disease. We explore data from [1], and ultimately provide a predictive linear model based on these variables. The data set includes information related to demographics, household compositon, social network, and a series of health metrics. The data also includes the value 'cogscore', which is an accumulative cognitive score based on two-word recall tests, two numeracy tests, and an orientation test for the study participants. We use 'cogscore' as a proxy for dementia risk and severity, with a higher 'cogscore' indicating lower cognitive impairment. Thus, our predictive model will aim to accuratetly predict 'cogscore' based on a number of selected features in the data set.

Before beginning the exploratory data analysis (EDA) and feature engineering phase of model construction, we combined the data in the initial easyshare.csv data set with the easyshare_all.csv data set by matching their rows on the values of the merge_id and wave columns, split the data into training and testing sets (choosing the size of the testing set to be 30% of the overall dataframe) and then resplitting these training and testing sets back into different dataframes based on whether their values came from easyshare.csv or easyshare_all.csv. This meant we had four datasets: d_train, d_test, extra_d_train, and extra_d_test. We did this in order to allow us to identify key variables in the additional easyshare_all.csv data set later on, which could aid the modelling process.

[20]:
```
# Immediately split dataframe into test and training sets so as to avoid data␣
↪leakage.

# Merge data, matched on 'mergeid' and 'wave'.
combined_d = pd.merge(original_d, extra_d, on=['mergeid', 'wave'])

# Fix a random seed for reproducible results.
seed = np.random.seed(42)

# Split into training and test data.
```

```
combined_d_train, combined_d_test = train_test_split(combined_d, test_size = 0.
 ↪3, random_state = seed)

# Split back into original and extra data for later merging of additional␣
 ↪variables.
n = original_d.shape[1]
d_train = (combined_d_train.iloc[:, :n]).copy()
d_test = (combined_d_test.iloc[:, :n]).copy()
extra_d_train = (combined_d_train.iloc[:, n:]).copy()
extra_d_test = (combined_d_test.iloc[:, n:]).copy()

# Rename columns to remove "_x" and "y" at the end of each column name after␣
 ↪combining.
d_train = d_train.rename(columns=lambda x: x.rstrip("_x") if x.endswith("_x")␣
 ↪else x)
d_test = d_test.rename(columns=lambda x: x.rstrip("_x") if x.endswith("_x")␣
 ↪else x)
extra_d_train = extra_d_train.rename(columns=lambda x: x.rstrip("_y") if x.
 ↪endswith("_y") else x)
extra_d_test = extra_d_test.rename(columns=lambda x: x.rstrip("_y") if x.
 ↪endswith("_y") else x)
```

3 Exploratory Data Analysis and Feature Engineering

To initiate the EDA process, we checked for NA values in the d_train dataset, and used the info and describe functions provided by the 'Pandas' python library to get a general overview of the data.

[21]:
```
# Check for presence of na values.
print(f'There are {d_train.isna().sum().sum()} NA values.')

# Get general overview of data.
d_train.info()
d_train.describe().round(2)

# Check if 'mergeid' is unique for every observation.
merge_id_count = d_train['mergeid'].nunique()
d_without_na_mergeid = d_train.dropna(subset=['mergeid'])

print(f"The 'mergeid' column has {merge_id_count} unique values.")
print(f"There are {(d_without_na_mergeid.shape[0])-merge_id_count} repeat␣
 ↪values of 'mergeid'.")
```

```
There are 47221 NA values.
<class 'pandas.core.frame.DataFrame'>
Int64Index: 68160 entries, 91211 to 15795
Data columns (total 23 columns):
```

```
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   mergeid         68160 non-null  object
 1   int_year        68160 non-null  float64
 2   wave            68160 non-null  float64
 3   country         68160 non-null  float64
 4   country_mod     68160 non-null  float64
 5   female          68160 non-null  float64
 6   age             68160 non-null  float64
 7   birth_country   68029 non-null  float64
 8   citizenship     68099 non-null  float64
 9   isced1997_r     68160 non-null  float64
 10  eduyears_mod    59083 non-null  float64
 11  eurod           67075 non-null  float64
 12  bmi             66281 non-null  float64
 13  bmi2            66281 non-null  float64
 14  smoking         67825 non-null  float64
 15  ever_smoked     67981 non-null  float64
 16  br010_mod       55380 non-null  float64
 17  br015           67980 non-null  float64
 18  casp            58842 non-null  float64
 19  chronic_mod     68095 non-null  float64
 20  sp008           58225 non-null  float64
 21  ch001           67843 non-null  float64
 22  cogscore        68160 non-null  float64
dtypes: float64(22), object(1)
memory usage: 12.5+ MB
The 'mergeid' column has 68160 unique values.
There are 0 repeat values of 'mergeid'.
```

As can be seen in the above outputs, we also investigated the 'mergeid' column of the dataset to determine if there were any repeat values. After seeing that the no such duplicates existed, as should be the case, we dropped the 'mergeid' column as it offered no useful information for the prediction of the 'cogscore' value.

[22]:
```
# Since mergeid is unique for every observation, we drop it.
d_train=d_train.drop("mergeid", axis = 1)
d_test=d_test.drop("mergeid", axis = 1)
```

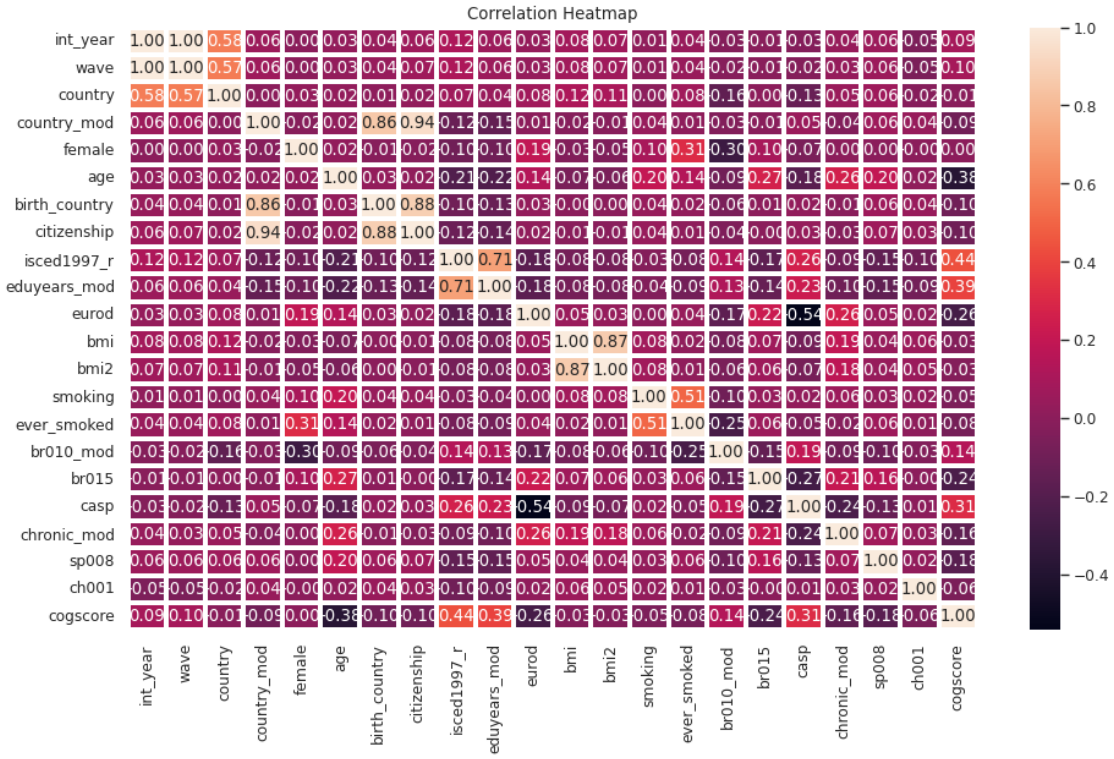
Next, we examined the distributions of each variable by plotting their histograms, and made use of pairplots and a correlation heatmap to examine the relationships between each pair of variables. The correlation heatmap showed significant correlations between various pairs of variables, including:

- eduyears_mod and isced1997_r
- citizenship and country_mod
- birth_country and country_mod
- citizenship and country_mod

- bmi and bmi2
- smoking and ever_smoked

We thus referred to the documentation [1] provided on the data to investigate the reasons behind these correlations, with the aim of determining whether certain variables should be dropped before fitting the model to reduce colinearity amongst the variables.

[23]:
```
# Plot heatmap of correlations.
sns.set(rc={'figure.figsize': (14, 8)})
sns.heatmap(d_train.corr(), annot = True, fmt = '.2f', linewidths = 2)
plt.title("Correlation Heatmap")
plt.show()
```



To begin with we looked at the variables eduyears_mod and isced1997_r. The easySHARE documentation [1] defines these variables as the years of education, and the ISCED coding of education, which classifies the subject's stage of education reached, respectively. Since both of these features describe the education level of the subject, we chose to remove one of them in order to prevent the effect of education on cognitive score being accounted for twice. Specifically, we chose to remove the eduyears_mod feature because its counterpart isced1997_r is categorized and therefore simpler to integrate into models. On top of this, isced1997_r showed a higher correlation with the target variable as can be seen from the heatmap.

[24]:
```
d_train = d_train.drop("eduyears_mod", axis = 1)
```

Next we investigated the three-way colinearity between the variables citizenship, birth_country, and country_mod. The easySHARE guide describes these variables as 'citizenship of respondent', 'country of birth', and 'modified country identifier' respectively. As we would expect, in many cases these three variables encoded the same information, with their differences primarily arising from migration or familial heritage. Specifically, in only 8% of instances was the birth_country value not equal to the country_mod value and in 10% was it not equal to the country_mod value.

[25]:
```
# Get number of instances where 'country_mod' is not equal to 'citizenship'.
not_equal_citizenship_count = (d_train['birth_country'] !=␣
 ↪d_train['citizenship']).sum()

# Get number of instances where 'country_mod' is not equal to 'birth_country'.
not_equal_country_mod_count = (d_train['birth_country'] !=␣
 ↪d_train['country_mod']).sum()

print(f"Percentage of instances where birth_country is not equal to citizenship:
 ↪ {not_equal_citizenship_count/len(d_train)}%")
print(f"Percentage of instances where birth_country is not equal to country_mod:
 ↪ {not_equal_country_mod_count/len(d_train)}%")
```

```
Percentage of instances where birth_country is not equal to citizenship:
0.08176349765258216%
Percentage of instances where birth_country is not equal to country_mod:
0.10177523474178404%
```

Moreover, from the heatmap, we see that birth_country and citizenship share a 0.88 correlation, birth_country and country_mod share a correlation of 0.86, and citizenship and country_mod share a correlation of 0.94. It is also worth noting that there is a further variable called country, however this variable encodes very similar information to country_mod (although the two are not correlated), differing only by whether or not we use the ISO code scheme or not. Additionally, in relation to the target variable cogscore these three features exhibited roughly equal correlations.

Given these findings, we decided to only keep the birth_country variable as it is likely more indicative of genetic factors, making it marginally more relevant for the prediction of dementia risk, and thus only this column was included in the selected_columns list we define later which includes the names of all the columns that were used in the model fitting.

Following on from the investigation of these variables, we identified the variables most highly correlated with cogscore to begin deciding which other factors to use in our predictive model.

[26]:
```
# Drop 'string' type objects from extra_d_train/test before calculating␣
 ↪numerical correlations.
extra_d_train = extra_d_train.select_dtypes(exclude=['object'])
extra_d_test = extra_d_test.select_dtypes(exclude=['object'])

# Add 'cogscore' to extra_d_train for correlation purposes.
extra_d_train['cogscore'] = d_train['cogscore']

# Calculate the correlation matrix for extra_d_train.
```

```
correlation_matrix = extra_d_train.corr()

# Isolate only the 'cogscore' correlations.
cogscore_correlations = correlation_matrix['cogscore']

# Exclude 'cogscore' as not interested in self-correlation.
cogscore_correlations = cogscore_correlations.drop(labels=['cogscore'])

# Sort and find the top ten most correlated variables to 'cogscore'.
top_correlations = cogscore_correlations.abs().nlargest(15)

print("Fifteen highest correlated variables with 'cogscore':")
print(top_correlations)
```

```
Fifteen highest correlated variables with 'cogscore':
recall_2       0.902263
recall_1       0.891473
isced1997_r    0.439779
dn003_mod      0.389272
age            0.384504
numeracy_1     0.346443
maxgrip        0.283442
mobilityind    0.274359
sphus          0.265371
grossmotor     0.255592
ep036_mod      0.252384
ep009_mod      0.249305
iadlza         0.240951
lgmuscle       0.236837
orienti        0.235103
Name: cogscore, dtype: float64
```

From the above results, we immediately saw that recall_1, recall_2 and numeracy_1 were highly correlated with cogscore. This of course was expected since these variables are 3 of the numbers that were directly used to compute cogscore. As such, despite their significant correlation to the target variable, we did not include them in our predictive model as they do not inform us of anything interesting. Also, dn003_mod is defined to be the year of birth of the subject, thus because this variable encodes the same information as age, we disregarded it.

From the data that was not already included in the easySHARE.csv data set, we noticed that maxgrip - the measure of maximum grip strength from 0 to 100, mobilityind - mobility index ranging from 0 to 4 (high: has difficulties), and sphus - self-perceived health (1 to 5, 5 being the worst) were the highest correlated variables with cogscore. Because of these high correlations with the target variable, and the widespread literature [2] which identifies these factors as significant predictors of dementia, we decided to include these extra variables from the easySHARE_all.csv data set in the model fitting process.

Upon plotting the three extra variables against cogscore, we noticed that they each contained invalid points. According to the easySHARE documentation, mobilityind scores should be between

0 and 4, maxgrip scores should be between 0 and 100, and sphus scores should be between 1 and 5, however the plots showed the existence of points outside these ranges. In particular, the data provided for sphus presented 62 instances in which the recorded value was outside of this range, there were many instances (5,108 in total) in which a negative maxgrip results were provided, and the mobilityind variable lay outside of the defined range (0 to 4) a total of 53 times. Whilst all of these instances were removed from the training dataset before fitting the model, many of them coincided, thus in total we only removed 5,178 instances of data as opposed to the combined sum of 5,108 + 62 + 53 = 5223 instances. These invalid values made up 5,178/68,160 = 7.6% of the overall dataset.

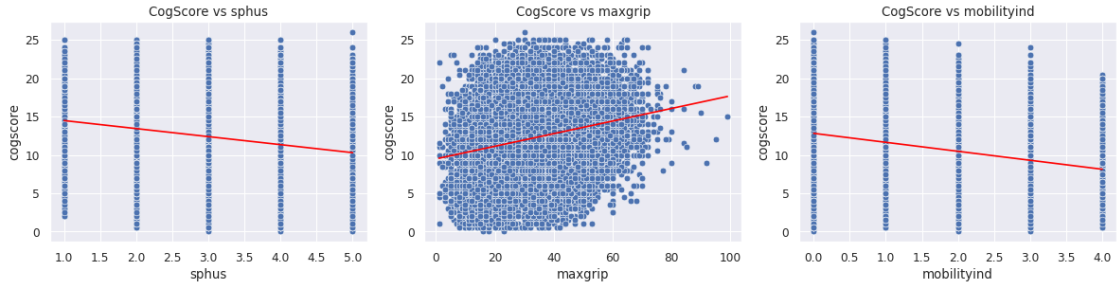
[28]:
```
# Check for invalid values in 'sphus', 'maxgrip' and 'mobilityind' columns.
maxgrip_invalid_count = extra_d_train[(extra_d_train.maxgrip < 0) |␣
 ↪(extra_d_train.maxgrip > 100)]['maxgrip'].shape[0]
sphus_invalid_count = extra_d_train[(extra_d_train.sphus < 1) | (extra_d_train.
 ↪sphus > 5)]['sphus'].shape[0]
mobility_invalid_count = extra_d_train[(extra_d_train.mobilityind < 0) |␣
 ↪(extra_d_train.mobilityind > 4)]['mobilityind'].shape[0]
overlapping_invalid_count = extra_d_train[(extra_d_train.sphus < 1) |␣
 ↪(extra_d_train.sphus > 5) |
                                          (extra_d_train.maxgrip < 0) |␣
 ↪(extra_d_train.maxgrip > 100) |
                                          (extra_d_train.mobilityind < 0) |␣
 ↪(extra_d_train.mobilityind > 4)].shape[0]

print(f"The 'maxgrip' column has {maxgrip_invalid_count} invalid inputs, the␣
 ↪'mobilityind' column has {mobility_invalid_count} invalid inputs, and the␣
 ↪'sphus' column has {sphus_invalid_count} invalid inputs")
print(f"In total there are {overlapping_invalid_count} instances of data with␣
 ↪invalid data inputs")
```

```
The 'maxgrip' column has 5108 invalid inputs, the 'mobilityind' column has 53
invalid inputs, and the 'sphus' column has 62 invalid inputs
In total there are 5178 instances of data with invalid data inputs
```

[29]:
```
# Clean the extra_d_train and extra_d_test datasets to remove invalid values␣
 ↪before looking at relationships
extra_d_train = extra_d_train[(extra_d_train.sphus >= 1) & (extra_d_train.sphus␣
 ↪<= 5)
                              & (extra_d_train.maxgrip >= 0) & (extra_d_train.
 ↪maxgrip <= 100)
                              & (extra_d_train.mobilityind >= 0) &␣
 ↪(extra_d_train.mobilityind <= 4)]

extra_d_test = extra_d_test[(extra_d_test.sphus >= 1) & (extra_d_test.sphus <=␣
 ↪5)
                            & (extra_d_test.maxgrip >= 0) & (extra_d_test.
 ↪maxgrip <= 100)
```

```
                                & (extra_d_test.mobilityind >= 0) &␣
↪(extra_d_test.mobilityind <= 4)]
```

[30]:
```
fig, axes = plt.subplots(nrows=1, ncols=len(extra_variables), figsize=(5 *␣
↪len(extra_variables), 4))

# Loop through the new variables create a scatter plot for each one.
for i, var in enumerate(extra_variables):
    sns.scatterplot(x=var, y='cogscore', data=extra_d_train, ax=axes[i])

     # Calculate coefficients for a line of best fit.
    coeffs = np.polyfit(extra_d_train[var], extra_d_train['cogscore'], deg=1)
    poly = np.poly1d(coeffs)

    # Generate x values from the min to max range of the variable.
    x_vals = np.linspace(extra_d_train[var].min(), extra_d_train[var].max(),␣
↪100)

    # Plot line of best fit.
    sns.lineplot(x=x_vals, y=poly(x_vals), ax=axes[i], color='red')

    axes[i].set_title(f'CogScore vs {var}')

plt.tight_layout()
plt.show()
```



Now, returning to the variables we identified as being highly correlated with each other, let us look at bmi and bmi2 more closely. Firstly, from the heatmap we saw that bmi and bmi2 had a correlation coefficient of 0.87. The former feature defines exactly what its name suggests, representing the body mass index of the subject, whereas, bmi2 categorises the bmi values into four distinct categories (1 to 4), with 1 indicating 'underweight' and 4 indicating 'obese'. Given this categorical representation of bmi2, which offers a simplified approach for modeling purposes, and considering its slightly stronger correlation with the target cogscore variable, we made the decision to exclude the bmi feature from further analysis.

The final pair of highly correlated variables which we identified were: smoking and ever_smoked. smoking indicates whether the subject currently smokes, while ever_smoked indicates if they have

ever smoked daily. Both variables had binary values: 1 for yes and 5 for no. Given that ever_smoked is not only more comprehensive but also exhibits a stronger correlation with the target variable (-0.08 as opposed to -0.05), we opted to drop the smoking variable.

Following all of the decisions outlined above, our finalised set of variables for modeling comprised of the following variables: int_year, female, age, birth_country, isced1997_r, eurod, bmi2, ever_smoked, br010_mod, br015, casp, chronic_mod, sp008, ch001, cogscore, sphus, maxgrip, and mobilityind. In particular, these were the names included in the previously mentioned selected_columns list. Note that we retained certain genetic features, such as birth_country and female to name a few, to determine those who are most at risk, emphasizing individuals to whom any policy recommendations are most significant.

[31]:
```
# Get the list of columns in d_train.
columns_in_d_train = d_train.columns.tolist()

# Drop any columns from extra_d_train and extra_d_test that are also in d_train␣
 ↪to avoid duplicates.
extra_d_train = extra_d_train[[col for col in extra_d_train.columns if col not␣
 ↪in columns_in_d_train]]
extra_d_test = extra_d_test[[col for col in extra_d_test.columns if col not in␣
 ↪columns_in_d_train]]

# Combine d_train and extra_d_train_filtered side by side.
combined_d_train = pd.concat([d_train, extra_d_train], axis=1)
combined_d_test = pd.concat([d_test, extra_d_test], axis=1)

# Specify list of columns to keep.
selected_columns = [
    'int_year', 'female', 'age', 'birth_country', 'isced1997_r', 'eurod',
    'bmi2', 'ever_smoked', 'br010_mod', 'br015', 'casp', 'chronic_mod', 'sp008',
    'ch001', 'cogscore', 'sphus', 'maxgrip', 'mobilityind'
]

# Proceed with our column selection for d_train and d_test.
d_train = combined_d_train[selected_columns]
d_test = combined_d_test[selected_columns]
```

One further factor which we considered when exploring the birth_country variable was that some respondents had registered uncommon values for this variable. These respondents had birth countries whose values made up less than 3% of the total dataset. In order to avoid potential challenges arising from countries appearing in the test set but not in the training set, we opted to group these countries into a single, unified category labeled '000.0'. Given that this variable is non-ordinal and categorical, this approach was deemed perfectly valid.

[33]:
```
# Count occurrences of each unique value in 'birth_country'
obscure_birth_country_counts_train = d_train['birth_country'].value_counts()
obscure_birth_country_counts_test = d_test['birth_country'].value_counts()
```

```
# Filter counts where occurrences are less than 3% of dataset.
threshold_train = 0.03*len(d_train)
obscure_birth_countries_train =␣
 ↪obscure_birth_country_counts_train[obscure_birth_country_counts_train <␣
 ↪threshold_train]
threshold_test = 0.03*len(d_test)

# Get the number of 'birth_country' values that are less than 3% of total␣
 ↪dataset.
num_obscure_birth_countries = obscure_birth_countries_train.count()

print(f"Number of 'birth_country' values that are less than 3% of total dataset:
 ↪ {num_obscure_birth_countries}")

# Identify countries to group into new other = 000.0 category.
countries_to_group_train =␣
 ↪obscure_birth_country_counts_train[obscure_birth_country_counts_train <␣
 ↪threshold_train].index
countries_to_group_test =␣
 ↪obscure_birth_country_counts_test[obscure_birth_country_counts_test <␣
 ↪threshold_test].index

# Replace 'birth_country' column with grouped values in both training and test␣
 ↪sets.
d_train['birth_country'] = d_train['birth_country'].apply(lambda x: 000.0 if x␣
 ↪in countries_to_group_train else x)
d_test['birth_country'] = d_test['birth_country'].apply(lambda x: 000.0 if x in␣
 ↪countries_to_group_test else x)
```

```
Number of 'birth_country' values that are less than 3% of total dataset: 142
```

The remaining dataframe after removing redundant variables, and performing the above transformations, is what we used in the model fitting process.

4 Model Fitting and Tuning

Following our EDA and prior to any model construction, we partitioned both the d_train and d_test dataframes (representing the training and test datasets, respectively) into feature and target arrays denoted as X_train, y_train, X_test, and y_test. In this partitioning, the 'y' arrays exclusively contained the cogscore column from the original d_train and d_test dataframes, while the 'X' arrays included all other columns.

After the feature and target variable arrays were created, we applied the sampler na_sampler taken from workshop 1 to these arrays. This procedure ensured the elimination of any NA values, meaning that our data was fully prepared for training the linear model.

[35]:
```
# Print the counts of na values and rows to ensure we know to deal with them␣
 ↪later.
d_no_na = original_d.dropna()
print(f'There are {original_d.duplicated().sum()} duplicated observations')
print(f'After removing na values there are {d_no_na.duplicated().sum()}␣
 ↪duplicated observations')
print(f'Thus there are {original_d.duplicated().sum()-d_no_na.duplicated().
 ↪sum()} rows of entirely NA values')
```

```
There are 975 duplicated observations
After removing na values there are 0 duplicated observations
Thus there are 975 rows of entirely NA values
```

[36]:
```
# Define function (from workshop 1) to drop na values, so we can use it for␣
 ↪pipelines.
def drop_na(X, y, axis = 0):

    data = np.concatenate([X,y.reshape(-1,1)], axis=1)
    df = pd.DataFrame(data)
    df = df.dropna(axis=axis)
    data = df.values

    return data[:,:-1], data[:,-1]

# Create sampler using function from workshop 1.
na_sampler = FunctionSampler(func=drop_na,
                             validate=False)
```

[37]:
```
# Fix a random seed for reproducible results.
seed = np.random.seed(0)

# Split training data into target variable and features.
X_train = d_train.drop("cogscore", axis = 1)
y_train = d_train["cogscore"].copy()
X_test = d_test.drop("cogscore", axis = 1)
y_test = d_test["cogscore"].copy()

# Separate feature names from values.
features = list(X_train.columns)

# Remove NA values.
X_train, y_train = na_sampler.fit_resample(X_train.values, y_train.values)
X_test, y_test = na_sampler.fit_resample(X_test.values, y_test.values)
```

4.0.1 BASELINE MODEL

To establish a baseline for comparison with the results of all of our refined models, we constructed a 'baseline linear model'. This model utilised only the features available in the simplified easySHARE.csv dataset, and employed a basic linear model without any further refinement or tuning. By doing so, we aimed to provide a benchmark against which the performance of our other models could be evaluated.

In order to make these comparisons we created a dataframe in which to store key model performance metrics, for all models tested. (As well as the models described in this section, we tested a variety of other models whose key performance metrics we also stored in this dataframe. This allowed us to justify our rejection of these other models, which we discuss in the 'Discussion & Conclusions' section of this report).

[38]:
```
# Create dataframe to store all model performance data.
column_names_for_df = ['Model', 'Train MSE', 'Train RMSE', 'Train R^2', 'Test␣
 ↪MSE', 'Test RMSE', 'Test R^2', '# Terms']
ModelData = pd.DataFrame(columns=column_names_for_df)
```

In order to fit this baseline model, we categorised our features into numerical, skewed numerical, categorical, and count features. Upon reviewing the documentation, we identified age and casp as numerical features. Additionally, the histograms revealed that casp exhibited a skewed distribution, leading us to classify it as a skewed numerical feature. Furthermore, chronic_mod and ch001 were identified as count features, all of the remaining features were deemed categorical. The rationale behind this categorization stemmed from the different preprocessing steps required for each category.

For the non-skewed numerical features, the preprocessing stage solely involved standardization, which was accomplished using the *StandardScaler* function. This function computes the standardised value for each data point by subtracting the mean of the entire feature and then dividing by the standard deviation. The transformation can be expressed as:

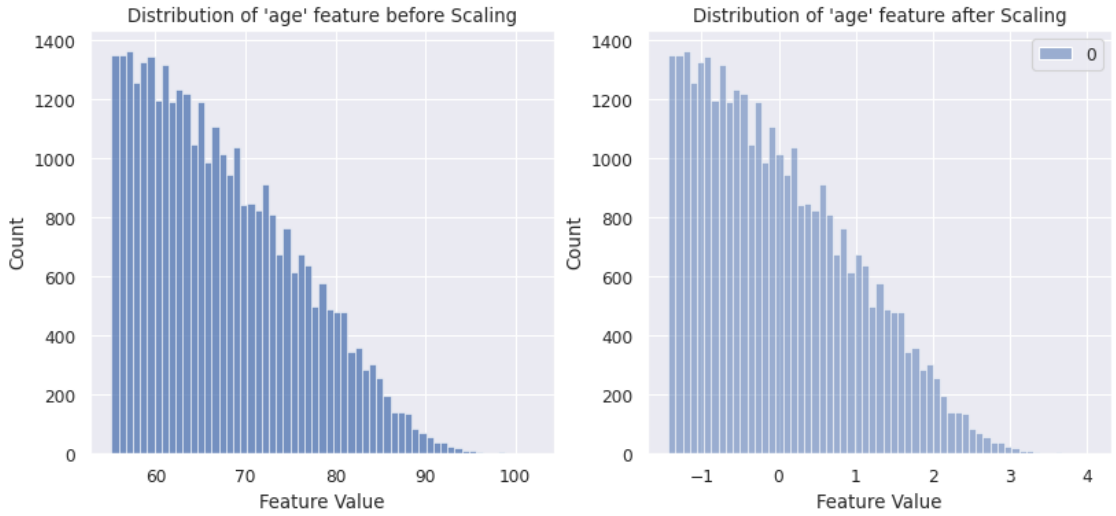
$$z_i = \frac{x_i - \bar{x}}{s},$$

where z_i represents the standardised value of the i^{th} input for the feature, $\bar{x}$ is the overall sample average for the feature, and s represents the observed standard deviation of the feature. This transformation ensures that the new standardised version of the feature possesses an overall sample mean of zero and unit sample variance. This is beneficial as it ensures that all features are on the same scale, and are directly comparable in terms of their influence on the model.

The plots below show the distribution of the age feature before and after this standard scaling was performed.

[39]:
```
fig, ax = plt.subplots(1,2, figsize=(12,5))
scaled_2 = StandardScaler().fit_transform(X_train[:,2].reshape(-1,1))

sns.histplot(data= X_train[:,2], ax = ax[0])
sns.histplot(data= scaled_2, ax = ax[1])
```

```
ax[0].set_title("Distribution of 'age' feature before Scaling")
ax[0].set_xlabel("Feature Value")
ax[1].set_title("Distribution of 'age' feature after Scaling")
ax[1].set_xlabel("Feature Value")

plt.show()
```



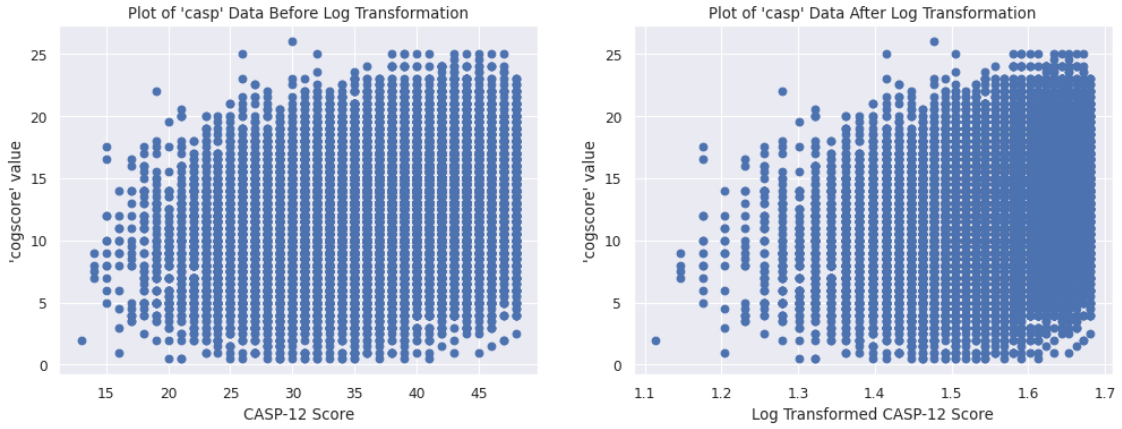
As we can see, the scaled version of the feature is distributed around zero and has a much smaller variance than the unscaled features.

For the skewed numerical features, we first applied a log transformation using the LogTransform() function and then applied a further standard scaling. The purpose of the log transformation of the skewed numerical features, which in this case was only casp which was defined as the CASP-12 quality of life score defined on a scale of 12-48, was to normalise the data since as we can see from the histogram of the original casp feature values, it was slightly skewed towards lower values.

This log transformation also helped to produce a more linear relationship with the target cogscore variable. As we can see from the plots below, there is a slightly more clearly defined positive relationship between the transformed version of the casp feature and the cogscore variable than between the original casp feature and cogscore.

[40]:
```
log_10 = LogTransformer(base='10').fit_transform(X_train[:,10].reshape(-1,1))

fig, ax = plt.subplots(1,2, figsize=(15,5))
ax[0].scatter(X_train[:,10].reshape(-1,1),y_train)
ax[0].set_title("Plot of 'casp' Data Before Log Transformation")
ax[0].set_xlabel("CASP-12 Score")
ax[0].set_ylabel("'cogscore' value")

ax[1].scatter(log_10, y_train)
ax[1].set_title("Plot of 'casp' Data After Log Transformation")
```

```
ax[1].set_xlabel("Log Transformed CASP-12 Score")
ax[1].set_ylabel("'cogscore' value")
plt.show()
```



In order to process the 'count' features we used a modified version of the boxcoxarray function defined in workshop 1. The original function is designed to identify and apply the optimal Box-Cox transformation for each feature. Our dataset included zero or negative values, which are incompatible with the standard Box-Cox transformation. To resolve this, we adapted the function to translate any non-positive values into the positive domain before applying the transformation. This adjustment does not impact the predictive capabilities of our model. However, it is important to consider this modification when interpreting the transformed features, as the shift to positive values can shift the meaning of the coefficients in terms of the original data. The purpose of this transformation was to give the features a more gaussian distribution which would improve the models performance by reducing its sensitivity to the scale and distribution of these count features.

Following this Box-Cox transformation, we applied a further standard scaling transformation to center these features and give them unit variance.

[41]:
```
# Define a shifting version of function (from workshop 1) for optimal box-cox␣
↪transformation.
def boxcoxarray(X, y=None, lmbda=None, shift=0):
    X_ = X.copy()
    # Shift values to handle zero or negative values.
    X_shifted = X_ + shift if shift else X_
    for i in range(X_shifted.shape[1]):
        # Apply Box-Cox transformation only on positive values.
        mask = X_shifted[:, i] > 0
        if np.any(mask):
            if lmbda == None:
                X_shifted[mask, i], bc_params_ = stats.boxcox(X_shifted[mask,␣
↪i])
            else:
```

```
                X_shifted[mask, i] = stats.boxcox(X_shifted[mask, i],␣
 ↪lmbda=lmbda)
    return X_shifted if shift else X_

# Create the shifting transformer with optimal lambda for pipeline.
BoxCoxShiftTransformer = FunctionTransformer(boxcoxarray, kw_args={'shift': 1},␣
 ↪validate=True)
```

For the categorical features we applied a one hot encoding of these variables, using the OneHotEncoder function provided by sklearn, before again standardising them using the StandardScaler function, however, this time setting the parameter 'with_mean=False'. We did this because after being one hot encoded the input data was sparse. For sparse input data, centering could destroy the sparseness property. When 'with_mean=False', the data is not centred and only the scale is adjusted, so the zero entries are maintained and the sparse structure of the matrix is not altered.

These pre-processing steps were then combined into an overall pipeline FinalPipeline_baseline which defined our baseline model.

[42]:
```
# Define indices of different variable types for pipeline.
numerical_indices_baseline = [2]
skewed_numerical_indices_baseline = [10]
categorical_indices_baseline = [0, 1, 3, 4, 5, 6, 7, 8, 9, 12]
count_indices_baseline = [11, 13]

# Define baseline processors.
numerical_processor_baseline = Pipeline([
    ("num_scale", StandardScaler())])

numerical_processor_baseline_2 = Pipeline([
    ("num_transform", LogTransformer()),
    ("num_scale", StandardScaler())])

count_processor_baseline = Pipeline([
    ("num_transform", BoxCoxShiftTransformer),
    ("num_scale", StandardScaler())])

categorical_processor_baseline = Pipeline([
    ("cat_encode", OneHotEncoder()),
    ("num_scale", StandardScaler(with_mean=False))])

# Define baseline ML pipeline.
FinalPipeline_baseline = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("numerical_processor", numerical_processor_baseline,␣
 ↪numerical_indices_baseline),
        ("numerical_processor_baseline_2", numerical_processor_baseline_2,␣
 ↪skewed_numerical_indices_baseline),
        ("count_processor", count_processor_baseline, count_indices_baseline),
```

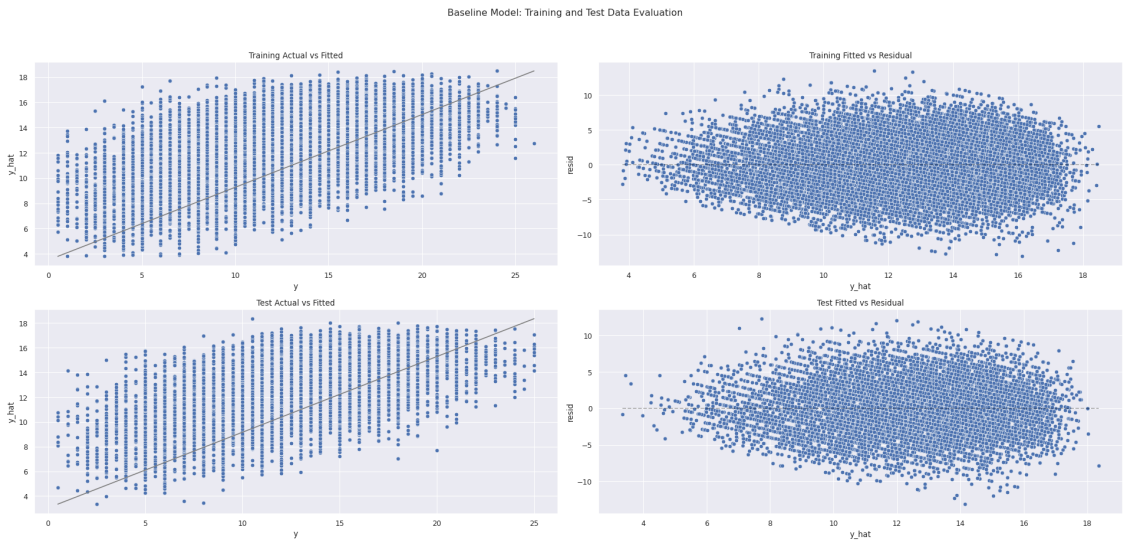
```
        ("categorical_processor", categorical_processor_baseline,␣
 ↪categorical_indices_baseline)])),
    ("model", LinearRegression())
])
```

[43]:
```
# Fit baseline model.
baseline_model = FinalPipeline_baseline.fit(X_train, y_train)

# Access the coefficients.
baseline_coefficients = (FinalPipeline_baseline.named_steps['model']).coef_

# Compute fitted values.
y_fit_baseline = baseline_model.predict(X_train)

# Compute predicted values.
y_pred_baseline = baseline_model.predict(X_test)

# Compute residuals for baseline model.
baseline_residual_frame_train = pd.DataFrame({'y': y_train, 'y_hat':␣
 ↪y_fit_baseline, 'resid': y_train - y_fit_baseline})
baseline_residual_frame_test = pd.DataFrame({'y': y_test, 'y_hat':␣
 ↪y_pred_baseline, 'resid': y_test - y_pred_baseline})

# Compute performance metrics.
M0TrainMSE = mean_squared_error(y_train, y_fit_baseline)
M0TrainRMSE = np.sqrt(M0TrainMSE)
M0TrainR2 = r2_score(y_train, y_fit_baseline)

M0TestMSE = mean_squared_error(y_test, y_pred_baseline)
M0TestRMSE = np.sqrt(M0TestMSE)
M0TestR2 = r2_score(y_test, y_pred_baseline)

# Append metrics to ModelPerformance dataframe.
model_0_performance = pd.DataFrame([['Baseline Linear Regression Model',␣
 ↪M0TrainMSE, M0TrainRMSE, M0TrainR2,
                                     M0TestMSE, M0TestRMSE, M0TestR2,␣
 ↪len(baseline_coefficients)]],
                                   columns=column_names_for_df)
ModelData = pd.concat([ModelData, model_0_performance], ignore_index=False)
```

Following the construction of our baseline model, we plotted the fitted values against the actual cogscore values alongside the fitted values against their corresponding residuals. This allowed us to verify the assumptions made in the model construction.

[44]:
```
plt.figure(figsize=(24, 12))

# Training data plots.
```

```
plt.subplot(221)
sns.lineplot(x='y', y='y_hat', color="grey", data=pd.DataFrame({'y':␣
 ↪[min(y_train), max(y_train)], 'y_hat': [min(y_fit_baseline),␣
 ↪max(y_fit_baseline)]}))
sns.scatterplot(x='y', y='y_hat', data=baseline_residual_frame_train).
 ↪set_title("Training Actual vs Fitted")

plt.subplot(222)
sns.scatterplot(x='y_hat', y='resid', data=baseline_residual_frame_train).
 ↪set_title("Training Fitted vs Residual")
plt.hlines(y=0, xmin=min(baseline_residual_frame_train['y_hat']),␣
 ↪xmax=max(baseline_residual_frame_train['y_hat']), linestyles='dashed',␣
 ↪alpha=0.3, colors="black")

# Test data plots.
plt.subplot(223)
sns.lineplot(x='y', y='y_hat', color="grey", data=pd.DataFrame({'y':␣
 ↪[min(y_test), max(y_test)], 'y_hat': [min(y_pred_baseline),␣
 ↪max(y_pred_baseline)]}))
sns.scatterplot(x='y', y='y_hat', data=baseline_residual_frame_test).
 ↪set_title("Test Actual vs Fitted")

plt.subplot(224)
sns.scatterplot(x='y_hat', y='resid', data=baseline_residual_frame_test).
 ↪set_title("Test Fitted vs Residual")
plt.hlines(y=0, xmin=min(baseline_residual_frame_test['y_hat']),␣
 ↪xmax=max(baseline_residual_frame_test['y_hat']), linestyles='dashed',␣
 ↪alpha=0.3, colors="black")

plt.suptitle("Baseline Model: Training and Test Data Evaluation")
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



The plots present the performance of the baseline regression model, on both training and test datasets:

Top-left (Training Actual vs Fitted): The points are scattered around the line, but follow the trend, indicating a reasonable fit.

Top-right (Training Fitted vs Residual): The residuals appear to be spread out evenly across the range of predictions without a clear pattern. The lack of obvious structure or pattern to the residuals suggests that the model does indeed follow the homoscedasticity assumption. However, we note that the variance of the residuals does seem to increase slightly with the fitted values.

Bottom-left (Test Actual vs Fitted): The spread of points is similar to that of the training data. This suggests that the model exhibits consistent performance on both datasets.

Bottom-right (Test Fitted vs Residual): In this plot the residuals are plotted against the predicted values. The distribution of the residuals is similar to the residuals belonging to the training set, with no clear pattern, indicating that model's assumptions may hold for the test data as well.

Overall, the baseline model seems to have relatively consistent residual patterns between the training and test sets, suggesting we can assume that our assumptions about homoskedacity and that the normal linear model assumption is true. However, there is still a slight increase in the variance of the residuals as the predicted values increase, indicating potential for model improvement.

4.0.2 Improved Linear Model

Once we'd verified the assumptions outlined above using the reduced set of variables from only the easySHARE.csv dataset, we proceeded to expand on this model by including the additional variables which were discussed in the EDA section; sphus, maxgrip, mobilityind. We used the same pipeline structure as in our initial baseline model, however, we did investigate the application of the OrdinalEncoder function provided by sklearn.

For the categorical features we tested an alternative pipeline whose application would depend on

whether the categorical feature being processed was an ordinal feature or not. For example, in the case where we were working with an ordinal feature such as sphus which ranked the subject self-perceived health on a scale on 1 to 5 with 1 being 'excellent' and 5 being 'poor', we applied an ordinal encoding transformation using the OrdinalEncoder function. In contrast if we were working with a non-ordinal feature such as the birth_country feature which was equal to each country's specific identifier, and thus did not involve any ordinal ranking of quality in the definition of these categories, we applied a one-hot encoding transformation using the OneHotEncoder function, also provided by sklearn. This created a feature for each different possible value these non-ordinal features could take. Again, after applying these encodings to the categories, we applied scaling to standardize the encoded features.

However, in order to determine whether the modelling process could be simplified without necessarily affecting the results we compared this to the original model (with the same structure as the baseline) in which all categories only had a OneHotEncoding applied to them and found that the results were fairly similar. In particular, the original model produced an R^2 prediction accuracy score of 0.353409 compared to the simplified model which produced an R^2 accuracy score of 0.357742, thus the simplified model even performed better. This may be due to the fact that many of the ordinal categorical features' inputs were already in numerical form and thus the application of a further OrdinalEncoder function was unnecessary. As a result our improved linear model followed the same pipeline structure as the original baseline linear model, however, with updated categorical indices which allowed for the inclusion of the aforementioned additional variables.

[45]:
```
# Define indices of different variable types, updated to include extra columns␣
 ↪for pipeline.
numerical_indices = [2, 15]
skewed_numerical_indices = [10]
categorical_indices = [0, 1, 3, 4, 5, 6, 7, 8, 9, 12, 14, 16]
count_indices = [11, 13]
```

[46]:
```
# Non-imputing Pipeline, including extra data (use this after dropping na␣
 ↪values with na_sampler).
# For non-skewed numerical variables.
numerical_processor = Pipeline([
    ("num_scale", StandardScaler())])

# For skewed numerical variables, i.e., whose distribution is not symmetrical.
numerical_processor_2 = Pipeline([
    ("num_transform", LogTransformer()),
    ("num_scale", StandardScaler())])

# For count variables.
count_processor = Pipeline([
    ("num_transform", BoxCoxShiftTransformer),
    ("num_scale", StandardScaler())])

# For categorical variables.
categorical_processor = Pipeline([
    ("cat_encode", OneHotEncoder()),
```

```
    ("num_scale", StandardScaler(with_mean=False))])

# Overall ML pipeline including all steps above.
FinalPipeline = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("numerical_processor", numerical_processor, numerical_indices),
        ("numerical_processor_2", numerical_processor_2,␣
 ↪skewed_numerical_indices),
        ("count_processor", count_processor, count_indices),
        ("categorical_processor", categorical_processor,␣
 ↪categorical_indices)])),
    ("model", LinearRegression())
])
```

[47]:
```
# Fit linear model and access the coefficients.
linear_model = FinalPipeline.fit(X_train,y_train)
linear_coefficients = (FinalPipeline.named_steps['model']).coef_

# Compute fitted values.
y_fit_linear = linear_model.predict(X_train)

# Predicted values.
y_pred_linear = linear_model.predict(X_test)

# Compute residuals for linear model.
linear_residual_frame_train = pd.DataFrame({'y': y_train, 'y_hat':␣
 ↪y_fit_linear, 'resid': y_train - y_fit_linear})
linear_residual_frame_test = pd.DataFrame({'y': y_test, 'y_hat': y_pred_linear,␣
 ↪'resid': y_test - y_pred_linear})

# Compute performance metrics.
M1TrainMSE = mean_squared_error(y_train, y_fit_linear)
M1TrainRMSE = np.sqrt(M1TrainMSE)
M1TrainR2 = r2_score(y_train, y_fit_linear)

M1TestMSE = mean_squared_error(y_test, y_pred_linear)
M1TestRMSE = np.sqrt(M1TestMSE)
M1TestR2 = r2_score(y_test, y_pred_linear)

# Append metrics to ModelPerformance dataframe.
model_1_performance = pd.DataFrame([['Linear Model', M1TrainMSE, M1TrainRMSE,␣
 ↪M1TrainR2, M1TestMSE, M1TestRMSE, M1TestR2, len(linear_coefficients)]],
                                   columns=column_names_for_df)
ModelData = pd.concat([ModelData, model_1_performance], ignore_index=False)
```

5 Discussion & Conclusions

After establishing a baseline linear model for comparison, we explored a variety of modeling techniques to improve our predictive accuracy. This array of models included: an Improved Linear Model, a Least Squares Model, a Model with Specific Pair Interactions, a Model with All Interactions, a Nonlinear Model, a Grid-Searched Polynomial Model, a Ridge Model, and a Lasso Model. These models were all generally straightforward to implement, requiring minimal specialised adjustments. However, we outline some specific steps that were taken below.

1. In the Least Squares Model, we avoided direct inclusion of non-ordinal categorical variables. Including these without proper encoding could result in misinterpretation, as they lack inherent numerical value or order.

2. When we began exploring specific pair interactions within our 'Model with Specific Pair Interactions', our goal was to look for potentially influencial interaction effects between variables. To do this, we focused on variables with strong linear relationships to cogscore, selecting those with high correlation coefficients (as identified from the heatmap). We aimed to pair these variables with others that, while not highly correlated with them (to avoid multicollinearity), could influence their relationship with cogscore. Additionally, we used our knowledge of the dataset to make predictions about how the association between cogscore and specific variables might change with the introduction of another variable. This led us to consider the interaction terms:

   - age* bmi2: As the impact of BMI on cognitive score may vary with age.
   - birth_country * isced1997_r: To explore if the impact of educational level on cognitive score varies across different countries.
   - casp * chronic_mod: To see if the impact of chronic conditions on cognitive score varies with psychological well-being.
   - isced1997_r * eurod: To explore if the impact of educational level on cognitive score varies dependent on the individual's depression levels.

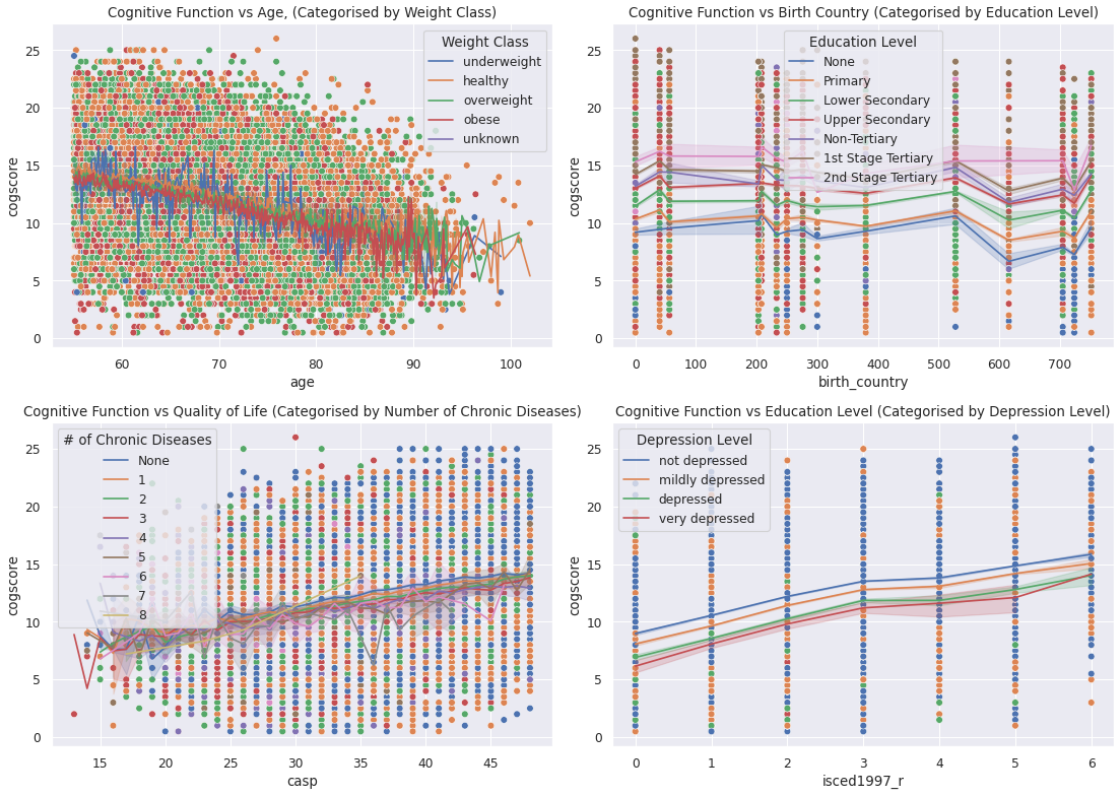
   To investigate these interactions, we plotted the first variable from each pair against cogscore, categorised by the second variable. This visual analysis helped us to identify whether the relationship between the interaction variables and cogscore was indeed significantly affected by the other variable in the interaction.

[48]:
```
# Define categories and ordered categories for each of the different plots.
# Weight Class Categories and Order for Legend (used for hue argument).
weight_categories = {1: 'underweight', 2: 'healthy', 3: 'overweight', 4:␣
 ↪'obese', 5: 'unknown'}
ordered_categories_1 = ['underweight', 'healthy', 'overweight', 'obese',␣
 ↪'unknown']
# Categorise the data points.
weight_vector = X_train[:, 6]
weight_class = [weight_categories[int(category)] for category in weight_vector]

# Education Level Categories and Order for Legend (used for hue argument).
education_levels = {
```

```
    0: 'None', 1: 'Primary', 2: 'Lower Secondary', 3: 'Upper Secondary', 4:␣
 ↪'Non-Tertiary',
    5: '1st Stage Tertiary', 6: '2nd Stage Tertiary'}
ordered_categories_2 = [
    'None', 'Primary', 'Lower Secondary', 'Upper Secondary', 'Non-Tertiary',
    '1st Stage Tertiary', '2nd Stage Tertiary']
education_vector = X_train[:, 4]
education_level = [education_levels[int(category)] for category in␣
 ↪education_vector]

# Number of Chronic Diseases Categories and Order for Legend (used for hue␣
 ↪argument).
num_diseases = {0: 'None', 1: '1', 2: '2', 3: '3', 4: '4', 5: '5', 6: '6', 7:␣
 ↪'7', 8: '8'}
ordered_categories_3 = [num_diseases[n] for n in sorted(num_diseases)]
disease_vector = X_train[:, 11]
num_disease = [num_diseases[int(category)] for category in disease_vector]

# Depression Level Categories.
depression_categories = {
    0: 'not depressed', 1: 'not depressed', 2: 'not depressed', 3: 'mildly␣
 ↪depressed',
    4: 'mildly depressed', 5: 'mildly depressed', 6: 'mildly depressed', 7:␣
 ↪'depressed',
    8: 'depressed', 9: 'depressed', 10: 'very depressed', 11: 'very depressed',␣
 ↪12: 'very depressed'
}
ordered_categories_4 = ['not depressed', 'mildly depressed', 'depressed', 'very␣
 ↪depressed']
depression_vector = X_train[:, 5]
depression_Class = [depression_categories[int(category)] for category in␣
 ↪depression_vector]
```

[49]:
```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
ax_flat = axes.flatten()

# Plot 1: Cognitive Function vs Age, (Categorised by Weight Class).
sns.scatterplot(ax=ax_flat[0], x=X_train[:, 2], y=y_train, hue=weight_class,␣
 ↪hue_order=ordered_categories_1, legend=False)
sns.lineplot(ax=ax_flat[0], x=X_train[:, 2], y=y_fit_linear, hue=weight_class,␣
 ↪hue_order=ordered_categories_1)
ax_flat[0].set_title('Cognitive Function vs Age, (Categorised by Weight Class)')
ax_flat[0].set_ylabel('cogscore')
ax_flat[0].set_xlabel('age')
ax_flat[0].legend(title='Weight Class')
```

```
# Plot 2: Cognitive Function vs Birth Country (Categorised by Education Level).
sns.scatterplot(ax=ax_flat[1], x=X_train[:, 3], y=y_train, hue=education_level,␣
 ↪hue_order=ordered_categories_2, legend=False)
sns.lineplot(ax=ax_flat[1], x=X_train[:, 3], y=y_fit_linear,␣
 ↪hue=education_level, hue_order=ordered_categories_2)
ax_flat[1].set_title('Cognitive Function vs Birth Country (Categorised by␣
 ↪Education Level)')
ax_flat[1].set_ylabel('cogscore')
ax_flat[1].set_xlabel('birth_country')
ax_flat[1].legend(title='Education Level')

# Plot 3: Cognitive Function vs Quality of Life (Categorised by Number of␣
 ↪Chronic Diseases).
sns.scatterplot(ax=ax_flat[2], x=X_train[:, 10], y=y_train, hue=num_disease,␣
 ↪hue_order=ordered_categories_3, legend=False)
sns.lineplot(ax=ax_flat[2], x=X_train[:, 10], y=y_fit_linear, hue=num_disease,␣
 ↪hue_order=ordered_categories_3)
ax_flat[2].set_title('Cognitive Function vs Quality of Life (Categorised by␣
 ↪Number of Chronic Diseases)')
ax_flat[2].set_ylabel('cogscore')
ax_flat[2].set_xlabel('casp')
ax_flat[2].legend(title='# of Chronic Diseases')

# Plot 4: Cognitive Function vs Education Level (Categorised by Depression␣
 ↪Level).
sns.scatterplot(ax=ax_flat[3], x=X_train[:, 4], y=y_train,␣
 ↪hue=depression_Class, hue_order=ordered_categories_4, legend=False)
sns.lineplot(ax=ax_flat[3], x=X_train[:, 4], y=y_fit_linear,␣
 ↪hue=depression_Class, hue_order=ordered_categories_4)
ax_flat[3].set_title('Cognitive Function vs Education Level (Categorised by␣
 ↪Depression Level)')
ax_flat[3].set_ylabel('cogscore')
ax_flat[3].set_xlabel('isced1997_r')
ax_flat[3].legend(title='Depression Level')

# Show final plot.
plt.tight_layout()
plt.show()
```



The presence of an interaction effect is suggested if the slope of the relationship between the first variable and cognitive score varies across the categories of the second variable.

The 'Cognitive Function vs Birth Country (Categorised by Education Level)' plot shows the spread of cognitive scores across different birth countries, categorised by education levels. Here we note minor differences in the shape of the slope of the relationship between individuals who have no education, i.e. answered 'none' and cognitive score, this suggests evidence of an interaction effect. Additionally, the "Cognitive Function vs Quality of Life (Categorized by Number of Chronic Diseases)" plot displays cognitive scores across different birth countries, again categorised by education levels. In this plot we note a minor difference in the trend for individuals with 3 cognitive diseases, suggesting another possible interaction effect.

To incorporate these observations into our model, we made specific adjustments to our dataset. First, we introduced a column to identify individuals with no prior education, taking on the value '1' if the individual has no previous education and 0 otherwise. Next, we created a column to represent the interaction between this new education variable and birth country. Similarly, for the observed trend among individuals with 3 cognitive diseases, we created an interaction column between having 3 cognitive diseases and 'casp'. We then ran this new updated data through the original linear regression pipeline, FinalPipeline.

3. With regards to the Lasso and Ridge models, appropriate feature engineering, including scaling, was carried out. For both the Ridge and Lasso regression models, StandardScaler()

was applied within their a pipelines to ensure that the features were properly scaled before applying regularisation. This was important as regularization penalizes the coefficients of the features, and without scaling, features with larger scales would be penalized more than those with smaller scales, regardless of their actual importance in predicting the target variable.

4. Furthermore, parameter tuning was done for both the Lasso and Ridge regression models using cross-validation approaches (LassoCV and RidgeCV for Lasso and Ridge, respectively). These methods automatically tune the regularization strength parameter (alpha) by trying out a range of values and selecting the one that minimizes cross-validation error. Additionally, the GridSearchCV function was used with a polynomial features pipeline to select the optimal degree for polynomial transformation, an approach employed to tune model hyperparameters and improve performance.

5. Finally it should also be noted that since Lasso regression is capable of performing feature selection due to its ability to shrink coefficients to zero, we included interaction terms in the features before applying Lasso. This allowed Lasso to select the most important interaction terms alongside the main effects. This was incorporated using the PolynomialFeatures with interaction_only=True parameter.

Revisiting the full set of models created, a key point was that we computed a set of performance metrics: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and the Coefficient of Determination (R^2) for both the training and testing datasets, as well as the count of terms included in the model. These statistics were recorded in a dataframe ModelData, allowing for an overview of all tested model's performance indicators. This table is shown below.

[120]:
```
ModelData
```

[120]:

	Model	Train MSE	Train RMSE	Train R^2	\
0	Baseline Linear Regression Model	10.411823	3.226736	0.347725	
1	Linear Model	10.310432	3.210986	0.354077	
2	Least Squares Model	10.893231	3.300490	0.317566	
3	Model with Specific Pair Interactions	10.295800	3.208707	0.354994	
4	Model with All Interactions	10.497818	3.240034	0.342338	
5	Nonlinear Model	10.293877	3.208407	0.355114	
6	Grid Searched Polynomial Model	10.537199	3.246105	0.339871	
7	Ridge Model	10.673042	3.266962	0.331361	
8	Lasso Model	10.566096	3.250553	0.338060	

	Test MSE	Test RMSE	Test R^2	# Terms
0	10.463778	3.234776	0.350276	69
1	10.343555	3.216140	0.357741	80
2	10.909837	3.303004	0.322579	15
3	10.331322	3.214237	0.358501	95
4	10.888309	3.299744	0.323916	861
5	10.318583	3.212255	0.359292	91
6	10.535778	3.245886	0.345805	84
7	10.638962	3.261742	0.339398	17
8	10.587392	3.253827	0.342601	153

The statistics shown in the ModelData dataframe reveal several key insights into the performance of our models.

1. **Baseline Linear Regression Model**: This model performs decently well in comparison to the other models across both datasets. However we do see improved performance metrics on later models suggesting that their are improvements to be made.
2. **Linear Model**: Showing a significant improvement over the baseline model, this model has lower MSE and RMSE values and higher R^2 values on both datasets. This suggests that the additional variables included in the linear model are beneficial for its performance, making it a better fit and more generalisable than the baseline.
3. **Least Squares Model**: This model exhibits weaker performance metrics than all of the other models. However it does have the advantage of far fewer coefficients, which makes the model simpler and more interpretable.
4. **Model with Specific Pair Interactions**: This model has a performance similar to the Linear Model, with a slightly lower MSE and RMSE on the test dataset and a slightly higher R^2 on both the train and test datasets compared to the Linear Model. This indicates that the specific interactions included may not add any significant predictive power beyond the linear terms, despite increasing the model's complexity.
5. **Model with All Interactions**: This model has a very high number of coefficients, and exhibits a drop in performance from training to testing, suggesting unecessary complexity and a potential for overfitting. Furthermore the increased complexity doesn't seem to provide a benefit over simpler models, as its test R^2 value is not among the highest.
6. **NonLinear Model**: It has the highest R^2 value on the test set, suggesting it may capture complex patterns in the data better than the linear models. However, we must acknowledge the risk of overfitting, as it has quite a high number of coefficients.
7. **Grid Searched Polynomial Model**: This model has a reasonably good Test R^2, indicating good generalisation. Its low Test RMSE suggests that its predictions are relatively close to the true values, on average.
8. **Ridge and Lasso Models**: The performance metrics of the Ridge and Lasso Models are quite similar to each other and slightly worse than the Linear Model. This might indicate that the data does not benefit significantly from regularisation, as these models don't outperform the simple linear model.

In order to choose the best model out of those listed above we considered several factors:

- **Generalisability**: The ability of the model to perform well on unseen data, indicated by the test metrics.
- **Performance**: Lower values of MSE and RMSE and higher values of R^2 on the test set.
- **Complexity**: The number of terms in the model, where a higher number can indicate greater model complexity, which could lead to overfitting.

Based on these considerations, we propose that the Linear Model provides the best balance between complexity and performance, with significantly improved metrics over the baseline and a reasonable number of terms. This lack of unnecessary complexity means it is computationally efficient and easily interpretable.

After selecting our chosen model, we began the process of interpretation. Initially, we retrieved the feature names directly from the model and transformed them into a format that was easily interpretable. Subsequently, we correlated these names with their respective coefficients from the Linear Model.

To allow for more insightful interpretations relevant to informed decision-making, we categorised the features into genetic and lifestyle groups. This categorisation allows for effectively tailoring recommendations and policy decisions. We then organised each group by the absolute value of their coefficients in descending order to identify the most influential features within each category.

[50]:
```
# List of original cateorical feature names.
categorical_features = [
    'int_year', 'female', 'birth_country', 'isced1997_r', 'eurod',
    'bmi2', 'ever_smoked', 'br010_mod', 'br015', 'sp008',
    'sphus', 'mobilityind'
]

# Access OneHotEncoder, which is part of a pipeline in the third transformer of␣
 ↪the ColumnTransformer.
one_hot_encoder = FinalPipeline.named_steps['pre_processing'].
 ↪transformers_[3][1].named_steps['cat_encode']

# Get feature names of the one hot encoded categorical features.
encoded_feature_names = one_hot_encoder.get_feature_names_out()

# Create a match between one hot encoded prefix and original feature name.
prefix_match = {f'x{i}_': f'{feature}_' for i, feature in␣
 ↪enumerate(categorical_features)}

# Replace the prefixes (x0, x1 etc.) in the encoded feature names with the␣
 ↪original feature names.
categorical_feature_names = [next((prefix_match[prefix] + name.split('_')[-1]␣
 ↪for prefix in prefix_match if name.startswith(prefix)), name) for name in␣
 ↪encoded_feature_names]

# List other types of feature names.
numerical_features = ['age']
skewed_numerical_feature_names = ['casp']
count_feature_names = ['chronic_mod', 'ch001']

# Join all feature names in the order specified by the model pipeline, and␣
 ↪include intercept.
all_feature_names = (['intercept'] + numerical_features +
                     skewed_numerical_feature_names +
                     count_feature_names +
                     categorical_feature_names)
```

```
# Create dataframe of feature names and their corresponding coeffecients.
feature_coefficients_df = pd.DataFrame({
    'Feature': all_feature_names,
    'Coefficient': linear_coefficients
})

# Remove intercept from dataframe before filtering for largest coefficients.
feature_coefficients_no_intercept_df = feature_coefficients_df[1:]

# Define a function to check if a feature name includes any of the strings from␣
 ↪a list.
def feature_includes_any_of(feature_name, feature_list):
    return any(feature_name.startswith(f"{feature}_") for feature in␣
 ↪feature_list)

# Specify the genetic and lifestyle features.
lifestyle_features = [
    'isced1997_r', 'eurod', 'bmi2', 'ever_smoked', 'br010_mod', 'br015',␣
 ↪'casp', 'sp008',
    'ch001', 'sphus', 'maxgrip', 'mobilityind'
]

genetic_features = [
    'int_year', 'female', 'age', 'birth_country', 'chronic_mod', 'maxgrip',␣
 ↪'mobilityind'
]

# Apply the function to filter lifestyle and genetic coefficients.
lifestyle_coefficients_df = feature_coefficients_no_intercept_df[
    feature_coefficients_no_intercept_df['Feature'].apply(lambda x:␣
 ↪feature_includes_any_of(x, lifestyle_features))]
genetic_coefficients_df = feature_coefficients_no_intercept_df[
    feature_coefficients_no_intercept_df['Feature'].apply(lambda x:␣
 ↪feature_includes_any_of(x, genetic_features))]

# Identify top coefficients by absolute value for each category and reset the␣
 ↪indices to match the ranking.
top_lifestyle_coefficients_df = lifestyle_coefficients_df.loc[
    lifestyle_coefficients_df.Coefficient.abs().nlargest(10).index
].reset_index(drop=True)
top_lifestyle_coefficients_df.index += 1
top_genetic_coefficients_df = genetic_coefficients_df.loc[
    genetic_coefficients_df.Coefficient.abs().nlargest(5).index
].reset_index(drop=True)
top_genetic_coefficients_df.index += 1
```

The predominant influential features from both categories were all categorical. Within our pro-

cessing pipeline, these features underwent one-hot encoding prior to being standardised using the StandardScaler function, specifically set with 'with_mean=False' to accommodate sparse data from the encoding. This simple preprocessing allowed for a straightforward translation of the coefficients back to the original data scale. To achieve this, we simply multiplied each coefficient by its corresponding standard deviation.

Since one-hot encoded features are binary (0 or 1), their standard deviation can be calculated from the training data after encoding. Since these variables take on values of 0 or 1, their variance (and thus standard deviation) reflects the proportion of $1s$ in the data. The formula for the standard deviation of a binary variable is:

$$\sqrt{p(1-p)}$$

where p is the proportion of data points that are 1 for a given feature.

Thus, in order to interpret the coefficients of these influential features in terms of the original data, we can calculate this quantity for any given feature and multiply it by its corresponding coefficient. We do this for each of the coefficients in the 'top_lifestyle_coefficients_df' and 'top_genetic_coefficients_df' dataframes.

[51]:
```
# Extract the value of the variable (third last position).
top_lifestyle_coefficients_df['Value'] =␣
 ↪(top_lifestyle_coefficients_df['Feature'].str[-3]).astype(int)

# Remove the last four characters from each feature name.
top_lifestyle_coefficients_df['Feature'] =␣
 ↪top_lifestyle_coefficients_df['Feature'].str[:-4]

top_lifestyle_coefficients_df['Adjusted Coefficient'] = 0
n = len(d_train)

# Iterate over each row in the top_lifestyle_coefficients_df dataframe.
for index, row in top_lifestyle_coefficients_df.iterrows():
    # The feature name from the current row.
    feature_name = row['Feature']
    # The value to count in the d_train dataframe.
    value_to_count = row['Value']
    # Count the occurrences of value in the crresponding d_train column.
    count = (d_train[feature_name] == value_to_count).sum()
    # Calculate proportion.
    p = count/n
    # Calculate standard deviation.
    sd = (p*(1-p))**(1/2)
    # Assign the count to the new column.
    top_lifestyle_coefficients_df.at[index, 'Adjusted Coefficient'] =␣
 ↪top_lifestyle_coefficients_df.at[index, 'Coefficient']*sd

# Apply a lambda function to the 'Feature' column to extract just the values.
```

```
top_genetic_coefficients_df['Value'] = top_genetic_coefficients_df['Feature'].
 ↪apply(
    lambda x: int(x.split('_')[-1][:-2]) if 'birth_country' in x
    else (int(x.split('_')[-1][:-2]) if x.startswith('int_year') else␣
 ↪int(x[-3])))

# Remove the last characters to get the original feature names.
top_genetic_coefficients_df['Feature'] = top_genetic_coefficients_df['Feature'].
 ↪apply(
    lambda x: x[:-6] if 'birth_country' in x else (x[:-7] if x.
 ↪startswith('int_year') else x[:-4]))

# Initialize a new column to store the adjusted coefficients.
top_genetic_coefficients_df['Adjusted Coefficient'] = 0

# Iterate over each row in the top_genetic_coefficients_df dataframe.
for index, row in top_genetic_coefficients_df.iterrows():

    # The feature name from the current row.
    feature_name = row['Feature']
    # The value to count in the d_train dataframe.
    value_to_count = row['Value']
    # Count the occurrences of value in the corresponding d_train column.
    count = (d_train[feature_name] == value_to_count).sum()
    # Calculate proportion.
    p = count / n
    # Calculate standard deviation.
    sd = (p * (1 - p)) ** 0.5
    # Assign the count to the new column.
    top_genetic_coefficients_df.at[index, 'Adjusted Coefficient'] =␣
 ↪top_genetic_coefficients_df.at[index, 'Coefficient'] * sd
```

[117]:
```
top_lifestyle_coefficients_df
```

[117]:

	Feature	Coefficient	Value	Adjusted Coefficient
1	isced1997_r	0.642611	5	0.248636
2	isced1997_r	-0.588339	1	-0.242983
3	isced1997_r	-0.412979	0	-0.090704
4	isced1997_r	0.212102	3	0.098699
5	isced1997_r	-0.199155	2	-0.076864
6	isced1997_r	0.198030	6	0.016439
7	isced1997_r	0.148347	4	0.029609
8	sphus	0.139138	2	0.050142
9	eurod	0.138864	0	0.056189
10	br010_mod	-0.114761	1	-0.051492

[118]:
```
top_genetic_coefficients_df
```

[118]:

	Feature	Coefficient	Value	Adjusted Coefficient
1	birth_country	-0.360803	724	-0.089539
2	female	-0.299979	0	-0.149459
3	female	0.299979	1	0.149459
4	int_year	-0.255108	2005	-0.055110
5	int_year	0.237439	2011	0.105566

After categorising factors into lifestyle and genetic factors, we analysed their correlations with cogscore. This allows us to make actionable decisions about suggesting policy.

The lifestyle factors that exhibited the strongest correlations (either positive or negative) with cogscore were:

- isced_1997_r interviewee's level of education ranging from isced_1997_r_0.0 = No education, to isced_1997_r_6.0 = Second stage tertiary education,
- sphus the self-perceived health of the interviewee ranging from 1 = excellent, to 5 = poor,
- eurod_ interviewee's self-perceived scale of depression ranging from 0 to 12,
- br010_mod interviewee's drinking behaviour ranging from 1 = Not at all, to 7 = Almost every day. Here br010_mod_1.0 corresponds to those interviewees who do not drink taking value 1.0 and those who do taking value 0.0.

We found that interviewees that had a greater level of education (upper secondary Education and above) suggest a positive correlation with cogscore. Interviewees with less education experience suggest a negative correlation with cogscore. Notice that the second stage of tertiary education isced1997_r_6.0 exhibited a considerably lower cogscore correlation than that of first-stage tertiary education isced_1997_r_5.0 which had the highest correlation with cogscore of the lifestyle factors. Furthermore, isced_1997_r_3.0 displayed a greater correlation than that of isced_1997_r_4.0, however, we note that there is a generally positive correlation with cogscore suggested by an increased level of education above upper secondary. We suggest that further research should be carried out to determine whether the marginal benefit of continuing education post secondary is pronounced in its correlation with cogscore.

Self-perceived health of the interviewees displayed suggests a positive correlation with cogscore independent of level of education of the interviewee and drinking behaviour. The negative correlation of br010_mod and cogscore suggests those who drink show increased values of cogscore.

The genetic factors that exhibited the strongest correlation (either positive or negative) with "cogscore" were:

- birth_country The country in which the interviewee was born,
- female_ where female_0.0 corresponds to the interviewees that were male and female_1.0 corresponds to those that were female,
- int_year The year in which the interview was held.

The adjusted coefficients reported by the genetic factors suggest that those interviewees who are female display stronger values of cogscore than those who are males.

The variables int_year suggest higher scores from the cognitive tests to calculate cogscore from the interviews held in 2011 and weaker scores in "2005". However, these are not necessarily indicitative

of any matieral correlations with cogscore but rather that the interviews held in 2011 were stronger by coincidence or some other underlying factors. Furthermore, the variable birth_country_724.0, corresponding to the interviewees born in Spain report a decrease in their cogscore of 0.089539.

Thus it can be inferred that for individuals possessing an education level represented by the encoding '5', there is a predicted increase of approximately 0.2486 in their cognitive score (cogscore), from this specific variable. The same procedure can be done for each of the other highly influential categorical features identified above.

Genetic factors suggest that males may be at a higher risk of developing dementia. Therefore, any general policy recommendations should prioritize addressing dementia risk factors specifically in males.

Regarding lifestyle factors, higher levels of education appear to be associated with a reduced risk of dementia. While it's important to acknowledge that individuals with higher education levels may perform better on cognitive tests regardless of dementia risk, dementia is a progressive disease that impacts cognitive function. Thus, even though there may be a correlation between education and cognitive scores, improving education levels should still help mitigate the effects of the disease. Additionally, the consumption of alcohol seems to increase the risk of dementia.

Furthermore, the inclusion of variables such as "sphus" (self-perceived health) and "eurod" (current depression levels) is noteworthy. This suggests that individuals experiencing depression or perceiving themselves as unhealthy should prioritize improving these aspects to potentially reduce their risk of dementia.

In summary, policy recommendations should focus on addressing dementia risk factors in males, promoting higher education levels, discouraging alcohol consumption, and emphasizing the importance of mental health and self-perceived health in dementia prevention efforts.

6 References

1. Börsch-Supan, A. & S. Gruber (2022): easySHARE. Release version: 8.0.0. SHARE-ERIC. Dataset. doi: 10.6103/SHARE.easy.800.

2. Cui M, Zhang S, Liu Y, Gang X, Wang G. Grip Strength and the Risk of Cognitive Decline and Dementia: A Systematic Review and Meta-Analysis of Longitudinal Cohort Studies. Front Aging Neurosci. 2021 Feb 4;13:625551. doi: 10.3389/fnagi.2021.625551. PMID: 33613270; PMCID: PMC7890203.