# *Practical 9*

*Jumping Rivers*

## *Question 1 - Advanced fuel economy*

Here we're going to follow on from the fuel economy question in practical 1. If you've lost the code, the following will get you back up to where you were

a) Yesterday we finished off by fitting the model $FE = \beta_0 + \beta_1 EngDispl + \beta_0 EngDispl^2$. Now we wish to add the transmission ('Tranmission') variable to our model. This variable is categorical so we will require some preprocessing prior to fitting the model. The following will create a column transformer which will standardise the numeric variables and one hot encode the categorical variable

```
x_train = np.hstack([
  X[['EngDispl']],
  X[['EngDispl']]*X[['EngDispl']],
  X[['Transmission']]
])
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

preprocessor = ColumnTransformer([
  ('num',StandardScaler(), [0,1]),
  ('cat',OneHotEncoder(), [2])
])
```

b) Create a pipeline that will run the preprocessor and fit a linear regression model

c) We can assess which model gave us the smallest overall mean squared error using the `mean_squared_error` function from the **sklearn.metrics** module.

```
from sklearn.metrics import mean_squared_error
```

d) The following code will grab you the fitted values from the model in the practical 2

Which model gave better performance?

## *Question 2 - diabetes*

For this practical we will explore models for the prediction of progression of diabetes for 442 patients. Measurements of their age, gender, body mass index, blood pressure and size blood serum measurements were taken to gether with a numeric measurement of disease progression one year after a baseline.

The data are available in the *jrpyanalytics* package and can be accessed with

```
import jrpyanalytics
diabetes = jrpyanalytics.datasets.diabetes.load_data()
```

The data have already been normalised, so we do not need to worry about this. However we should separate the inputs from the output ready for modelling.

```
X, y = diabetes.drop('y', axis=1), diabetes['y']
```

a) It is good practice to have a dedicated test set for final assessment of our chosen models. We can create training and test sets from data using `sklearn.model_selection.train_test_split()`. The following code will partition our data with 10% held out for final testing. The other 90% we will use for training and cross validation of different models.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
  X,y, test_size = 0.1,
  random_state = 2019, # ensures same random subset
)
```

b) Begin by fitting a linear regression to the training set using all available predictor variables.

c) Use the `mean_squared_error` function from the *sklearn.metrics* module on the full training set. This will give us the training error.

d) Training error gives us a measure of how far from the original data our model is. However it is typically different to test error, which would give us a better idea of how our model generalises to new data. Use 10 fold cross validation to estimate the test error rate for this model.

e) How does this compare to the training error?

f) The following code will return one bootstrapped estimate of the test error for our data.

```
from sklearn.utils import resample
from sklearn.preprocessing import StandardScaler
```

```
model = Pipeline(
  steps = [
    ('pre', StandardScaler()),
    ('reg', LinearRegression())
  ]
)
boot_X_train = resample(X, n_samples = X.shape[0], replace = True)
index = boot_X_train.index
boot_y_train = y[index]
not_index = [not i in index for i in X.index]
boot_X_test = X.iloc[not_index]
boot_y_test = y[not_index]
model.fit(boot_X_train, boot_y_train)

boot_train_pred = model.predict(boot_X_train)
boot_test_pred = model.predict(boot_X_test)
test_rmse = mean_squared_error(boot_y_test, boot_test_pred)
train_rmse = mean_squared_error(boot_y_train, boot_train_pred)
```

Create a for loop that will return 100 estimates.

g) Create a distribution plot of both the training and test RMSES using **seaborn**

```
import seaborn as sns
sns.distplot(boot_train_rmses)
sns.distplot(boot_test_rmses)
```

Does this back up your answer to question e)?