`passenger_forecasting`
Code Guide

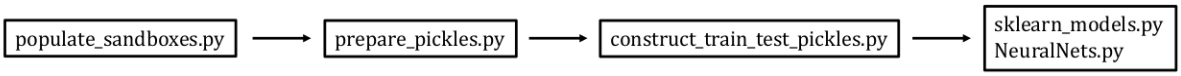
Jack Whaley-Baldwin

September 2021

The `passenger_forecasting` Package - Overview

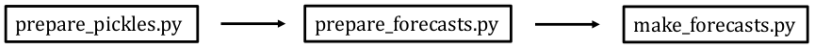
`passenger_forecasting` is a complete python package, designed to cover the entire process from pulling relevant data in the ICW through to modelling the data with various models.

The design philosophy of `passenger_forecasting` has been to automate as much as possible; getting from data-pull to forecast in as few commands as possible. In fact, the entire workflow to train and test a model is simply:



In a nutshell, `populate_sandboxes.py` adds relevant snapshots to the sandbox tables, from which one or more `.csv` files for the relevant train/testing period(s) can be created. These `.csv` files are then given to `prepare_pickles.py`, which merges, cleans and prepares the data, and saves it in the binary `.pkl` format. `construct_train_test_pickles.py` then takes this cleaned dataset, and prepares training and test sets for the ML models, also saving them in the `.pkl` format. Finally, these train and test sets are fed into either `sklearn_models.py` or `NeuralNets.py`, which will train and test a model on the supplied data, before saving the model to disk for use with future forecasting operations.

Once the above has been carried out and you're happy with the performance of the trained model, the forecasting workflow is simply:



Again, one or more `.csv` files are fed to `prepare_pickles.py`, but this time, this data represents *the flights to be forecasted for*. The `.pkl` file produced by `prepare_pickles.py` is fed into `prepare_forecasts.py`, which does a similar job to `construct_train_test_pickles.py`, with the important exception that this time, the data is not being used to train/test a model, but rather to make actual 'real-world' predictions. Finally, `make_forecasts.py` will use the model of your choice to produce seat factor predictions for all the flights in the supplied dataset, and will write these predictions to a `.csv` file.

Directory Structure

The directory structure of the `passenger_forecasting` package is shown below.

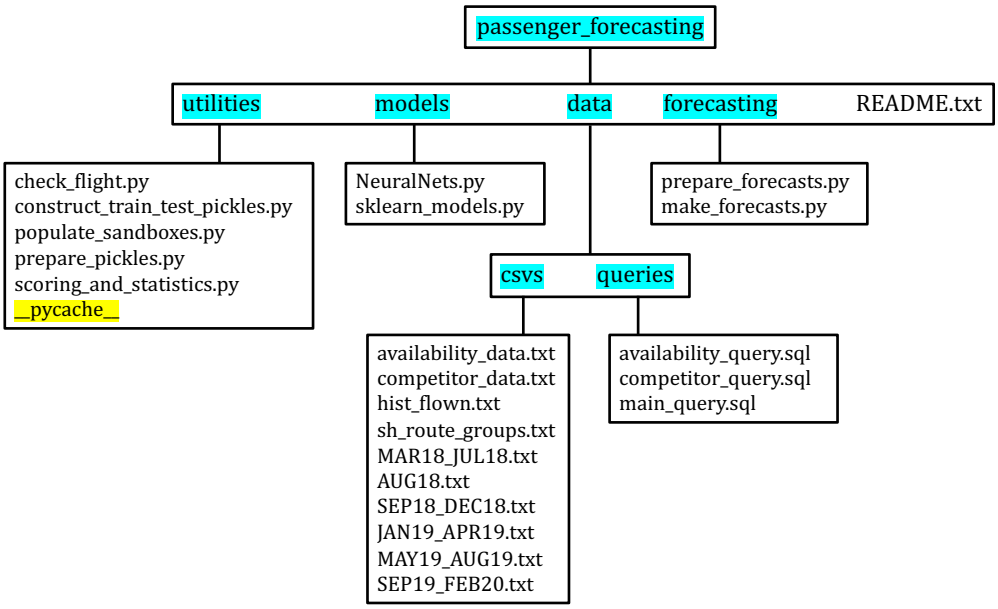


Figure 1: Directory structure of the passenger forecasting package. Elements highlighted in cyan are directories, and the __pycache__ directory (highlighted in yellow) may or may not be present, depending on whether you have run the code before (it will contain compiled python bytecode `.pyc` files). Everything else is a file.

Workflow - Data Preparation & Model Training/Testing

The schematic below is a more detailed version of the simplified data-prep and model training workflow shown at the start of this document.

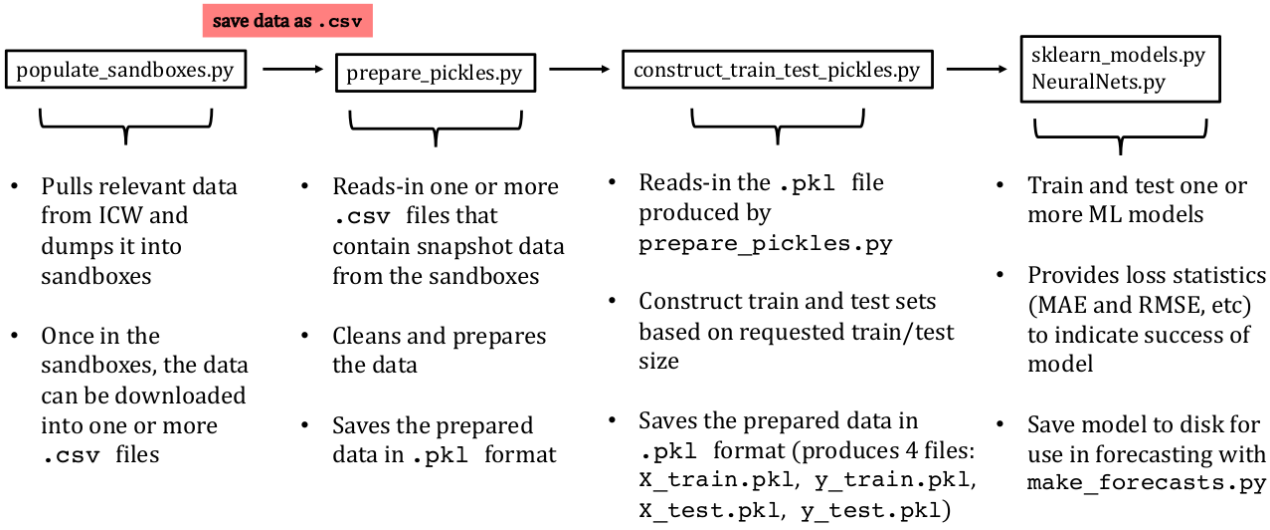


Figure 2: The workflow comprising data preparation & cleaning, production of train/test sets and model training and testing using the `passenger_forecasting` package. Detailed usage of each script is described on the following pages by means of a fully worked example.

Workflow - Producing Seat Factor Forecasts

The schematic below is a more detailed version of the simplified forecasting workflow shown at the start of this document.

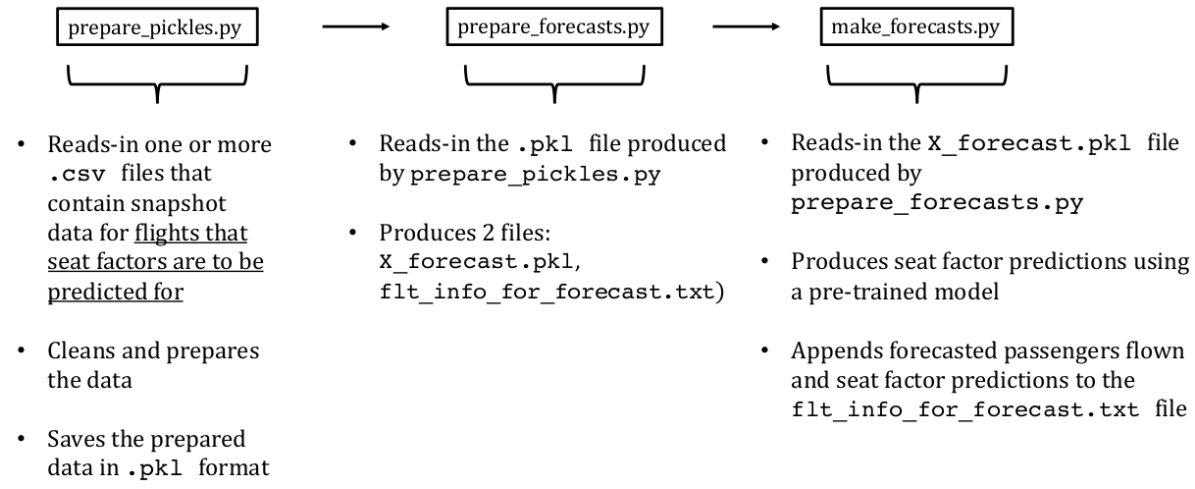


Figure 3: The workflow comprising the forecasting stage of the `passenger_forecasting` package. Detailed usage of each script is described on the following pages by means of a fully worked example.

Worked Example

Here, we will work through an example from scratch, although the use of the `populate_sandboxes.py` script will be skipped, as the sandboxes are already populated (or, at least they were when I last checked). The entirety of the data from these sandboxes, which spans March 2018 to February 2020, has already been downloaded in `.csv` format and can be found (split into multiple files) in the `data\csvs` directory. If it is desired that future data be added, they should be compliant in format with the csvs in this directory. Obviously, for flights yet to take place, the `flown_pax`, `op_flown_pax`, and `FLOWN_CAPACITY` columns are not yet known (in fact, we're trying to predict the former) - In this case these columns can be filled with dummy values (e.g. 999) as the forecasting part of the code will just ignore them.

It is assumed that you have already downloaded the package and that it lies in an accessible location, which we will refer to as `<package_directory>` . You will also need the `pandas`, `keras`, `sklearn`, `numpy`, `scipy` and `matplotlib` libraries installed.

In this worked example, we'll simulate a real-world forecasting scenario. We will use data from September 2018 up to August 2019 to train and test a few ML models, and then make seat factor predictions for flights in mid-late September 2019. This simulates the real-world situation where we train and test the model on as much data that we can (i.e. 'everything up until now'), and 'deploy' it on unseen data (in this case, the data in September 2019, which we'll forecast for).

We will aim to make these seat factor predictions 21 days out (21 DTD), using only weekly snapshots (that is; snapshot information taken every 7 DTD - 56,49,42,35,...,21). I'll use red for terminal input and blue for terminal output.

Before we start, I will mention that nearly all of the scripts we will use here can be run with a `--help` argument, which should provide useful info on expected input, code usage, etc.

Create a new folder (directory) somewhere convenient (perhaps call it `model_prep` or something like that). Copy the `SEP18_DEC18.txt`, `JAN19_APR19.txt` and `MAY19_AUG19.txt` files to this directory. These files comprise the raw data in `.csv` format as downloaded using (for example) TeradataSQL, and can all be found in `<package_directory>\data\csvs` . This will serve as our training and testing dataset.

Now, in the same directory as where you've copied the files, run the `prepare_pickles.py` script, which will read-in, merge and clean all of this data, producing a single dataset:

```
python <package_directory>\utilities\prepare_pickles.py SEP18_DEC18.txt
JAN19_APR19.txt MAY19_AUG19.txt -sampling_frequency 7
```

(the command is long because we've specified three files, and so it spills over onto the next line - it may or may not fit into one line on your terminal). Note that specifying `-sampling_frequency 7` tells the script to only pick data from snapshots every 7 days; that is, 7,14,21,28,35... etc. It actually happens that I've set the default value of `sampling_frequency` to 7 anyway (i.e. a snapshot every week), so it wasn't necessary to specify it in this case, but we did so anyway to be clear. Other alternatives are possible of course, e.g. `-sampling_frequency 3` which will sample 7,10,13,16,19,... etc. From experience, sampling every 7 days seems to be more than sufficient.

This should take around 3 minutes, and when complete, `prepare_pickles.py` will have produced a file called `CLEANED.pkl` in the current directory. This is a cleaned serialised dataset ready to be turned into train and test sets. The advantage is that the `.pkl` format can be rapidly read by `pandas`, and this DataFrame of hundreds of thousands of flights will now take just seconds to read in the future.

Before we construct the train and test sets, you may wish to do a sense check on the `CLEANED.pkl` file (that is; confirm it contains at least some of the flights you're expecting, sampled appropriately). To do this, I've developed the `check_flight.py` script, which reads `CLEANED.pkl` and plots summary statistics for a given flight, e.g. Flight 780 (LHR to ARN) departing on the 26th June 2019:

```
python <package_directory>\utilities\check_flight.py CLEANED.pkl
-flight_number 780 -departure_date 2019-06-26
```

(again, this may go across more than one line on your terminal). This should plot some summary statistics for this flight, and is a nice visualisation of the data that the ML models are going to receive:

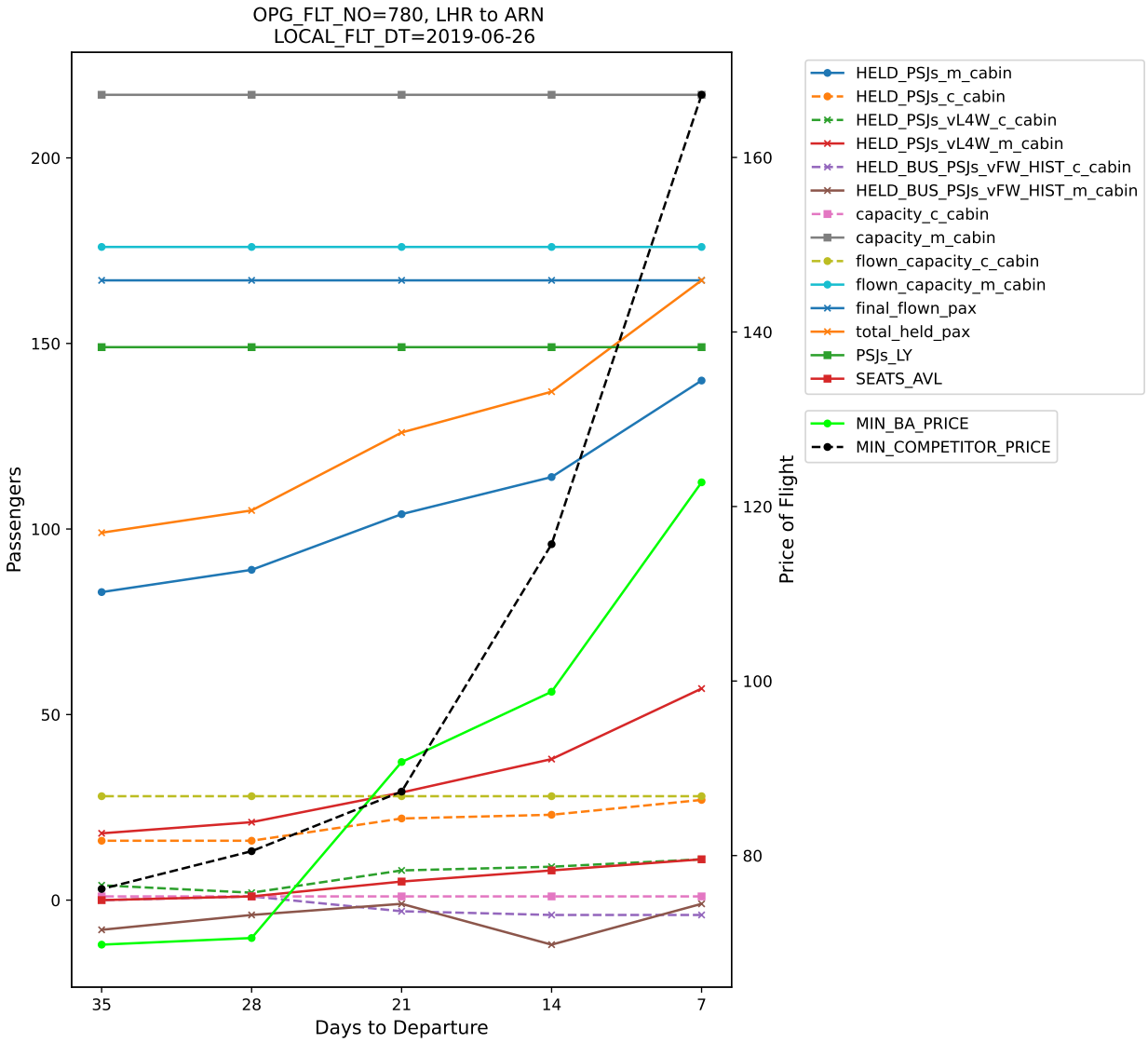


Figure 4: An example of the summary statistics plotted by the the `check_flights.py` script, in this case for flight 780 from LHR to ARN, departing on the 26th June 2019.

This plots a fair bit of information, so you may need to expand it to full screen to see everything. Once you've finished experimenting with `check_flight.py`, it's time to make the train and test sets. To achieve this, we use the `construct_train_test_pickles.py` script, also located in `<package_directory>\utilities`. For demonstration

purposes, let's construct a train set comprising 5000 flights and a test set comprising 1000 flights:

```
python <package_directory>\utilities\construct_train_test_pickles.py
CLEANED.pkl -train_size 5000 -test_size 1000 -forecast_DTD 21
```

This should take less than 2 minutes. The `-forecast_DTD 21` argument tells the code to only use those DTDs 21 days out and further (i.e. 21,28,35 etc) - The code will thus throw away DTDs 7 and 14 in our example here. When complete, there should now be four extra files in the directory: `X_train.pkl`, `y_train.pkl`, `X_test.pkl`, `y_test.pkl` - These represent the input-output pairs for the train set and test set respectively. The script will also print out:

```
DATA SHAPES after processing
X_train has shape: (5000, 105)
y_train has shape: (5000, 1)
X_test has shape: (1000, 105)
y_test has shape: (1000, 1)
```

Which tells us that it has indeed created 5000 training datapoints and 1000 testing datapoints, and that each input vector into the model has a length of 105 (see the `structure_of_input_vectors.pdf` file for an explanation of why, in this case, it's 105 elements long). Each output vector has a length of just 1, since we're predicting a single number every time (flown passengers). We're now ready to do some modelling.

For this demonstration, we will first use the models in the `sklearn_models.py` script, and then use the neural network model in the `NeuralNets.py` script later. First, let's run a simple Linear Regression model on the train/test set that we have just generated:

```
python <package_directory>\models\sklearn_models.py LinearRegression
-save_model_to_file
```

Note that we pass the `-save_model_to_file` flag, which will save the trained model to disk so we can use it later to make forecasts. This should take just a few seconds to run, and the script will print some useful statistics to the terminal (such as the coefficient of determination (R^2), MAE and RMSE). The script will also choose several flights randomly from the test set and print their prediction alongside the actual value. Extracting just part of the terminal printout, we get:

```
Using model: LinearRegression
R^2 on train set =  0.8113310978671937
R^2 on test set =  0.7984545786146127
LinearRegression avg. train set RMSE = 13.935078
LinearRegression avg. train set MAE = 10.579200
```

```
LinearRegression stdev of train set predictions = 28.827470
Actual stdev of train set = 32.025857
LinearRegression avg. test set RMSE = 13.494666
LinearRegression avg. test set MAE = 10.540000
LinearRegression stdev of test set predictions = 27.893251
Actual stdev of test set = 29.745654
Here are 15 randomly chosen predictions for LinearRegression...
y_pred= 122.000000, y_actual=130.000000
y_pred= 123.000000, y_actual=139.000000
y_pred= 125.000000, y_actual=125.000000
(...)
On these 15 randomly chosen predictions, LinearRegression achieved
a MAE of 16.933333 and a RMSE of 21.494185
Saved trained model to file 'LinearRegression.pkl'
RUN TIME: Took 0.773767 seconds to train and test model
```

Where I've copied only 3 of the 15 randomly chosen predictions here for conciseness. Your results may be slightly different since the train / test points are chosen at random. Note that the code also tells us that it has successfully saved the model to disk, and a `LinearRegression.pkl` file should have appeared in the current directory. As can be seen, 21 days out from departure, the Linear Regression model appears to be getting a mean absolute error (MAE) of around 10.5 on the test set, and a root-mean-squared error (RMSE) of around 13.5 on the test set - This is already pretty good. It is worth noting that in some rare cases, linear regression can give nonsensical results:

```
Using model: LinearRegression
R^2 on train set =  0.8222558695204466
R^2 on test set =  -3.1654615897381536e+21
LinearRegression avg. train set RMSE = 13.733383
LinearRegression avg. train set MAE = 10.540200
LinearRegression stdev of train set predictions = 29.659089
Actual stdev of train set = 32.548797
LinearRegression avg. test set RMSE = 1620953462125.765625
LinearRegression avg. test set MAE = 51259049224.788002
LinearRegression stdev of test set predictions = 1620142782673.906250
```

Note the enormous test MAE, RMSE and standard deviation, as well as the nonsensical R^2 value on the test set, despite the fact that the train set errors are perfectly reasonable. This simply means that the linear regression model has been unfortunate, and has been given a poor training set to which it has overfit. From my experience, this tends to happen about once in every 50 runs of the linear regression model. To fix this, simply generate a fresh dataset using `construct_train_test_pickles.py` as described earlier - Since train and test datapoints are picked at random, you'll get a different dataset this time.

Now, let's try a Random Forest regression model:

```
python <package_directory>\models\sklearn_models.py RandomForest -save_model_to_file
```

This will take slightly longer, probably around 20 seconds. The script should produce a similar printout to that for the Linear Regression model:

```
Using model: RandomForest
R^2 on train set =  0.9768907686545314
R^2 on test set =  0.7698346723288839
RandomForest avg. train set RMSE = 4.902448
RandomForest avg. train set MAE = 3.423200
RandomForest stdev of train set predictions = 30.102434
Actual stdev of train set = 32.025857
RandomForest avg. test set RMSE = 14.441018
RandomForest avg. test set MAE = 11.059000
RandomForest stdev of test set predictions = 26.876421
Actual stdev of test set = 29.745654
Here are 15 randomly chosen predictions for RandomForestRegressor...
y_pred= 115.000000, y_actual=119.000000
y_pred= 117.000000, y_actual=110.000000
y_pred= 107.000000, y_actual=123.000000
(...)
On these 15 randomly chosen predictions, RandomForestRegressor achieved
a MAE of 11.933333 and a RMSE of 14.946572
Saved trained model to file 'RandomForest.pkl'
RUN TIME: Took 15.445863 seconds to train and test model
```

Where I've again copied only 3 of the 15 randomly chosen predictions to save space. If you receive a warning from `sklearn` about the shape of the output, you may safely ignore this. Again, we're told the model has been successfully written to disk, and a corresponding `RandomForest.pkl` should have appeared in the directory. The Random Forest Regressor also scores very well, achieving a MAE of 11.1 and RMSE of 14.4 on the test set. Note the near perfect coefficient of determination (~ 0.977) and extremely small error (MAE ~ 3.4) on the train set - This is characteristic of Random Forest, and the 'real' performance of a RF model should be assessed on the test set only (see here).

You can also try a Ridge Regression (I won't bother writing this to disk, but feel free to do so):

```
python <package_directory>\models\sklearn_models.py Ridge
```

And a LinearSVR (again, I won't write this to disk):

```
python <package_directory>\models\sklearn_models.py LinearSVR
```

In fact, all of the models will perform quite well, due to the great explanatory depth of the set of input variables.

Finally, lets try training a neural network. Since we use neural networks implemented in `keras` (*à la* Tensorflow) the neural network model is not found in the `sklearn_models.py` script but rather in a separate script called `NeuralNets.py` (also located in the `models` directory). To train a neural net, simply type:

```
python <package_directory>\models\NeuralNets.py -epochs 125 -save_model_to_file
```

This will train a Neural Network with a single hidden layer containing 48 nodes (the default). Note that the default value for the number of training epochs is 250, but for this demonstration, we specify `-epochs 125` instead, which should be more than sufficient. As the model trains, loss metrics and training progress will be printed to the terminal. When finished, You should get a printout like:

```
METRIC LABELS:
['loss', 'mean_squared_error', 'mean_absolute_error']
NN train set metrics:
157/157 [==============================] - 0s 743us/step - loss: 200.7394
- mean_squared_error: 200.6809 - mean_absolute_error: 10.7345
[200.73941040039062, 200.68092346191406, 10.734530448913574]
NN test set metrics:
32/32 [==============================] - 0s 504us/step - loss: 178.3735
- mean_squared_error: 176.5774 - mean_absolute_error: 10.2915
[178.37353515625, 176.57736206054688, 10.29154109954834]
Here are 15 randomly chosen predictions for <keras.engine.sequential.Sequential
object at 0x0000023989FD1CD0>...
y_pred= 190.000000, y_actual=182.000000
y_pred= 150.000000, y_actual=146.000000
y_pred= 133.000000, y_actual=119.000000
(...)
On these 15 randomly chosen predictions, <keras.engine.sequential.Sequential
object at 0x0000023989FD1CD0> achieved a MAE of 9.533333 and a RMSE
of 12.173742
RUN TIME: Took 152.471180 seconds to train and test model
```

The format of the printout is slightly different to the other models, but the information provided is essentially the same (MAE, RMSE etc). As we can see, we get an MAE of 10.3 on the test set. The script also plots a train/test curve:

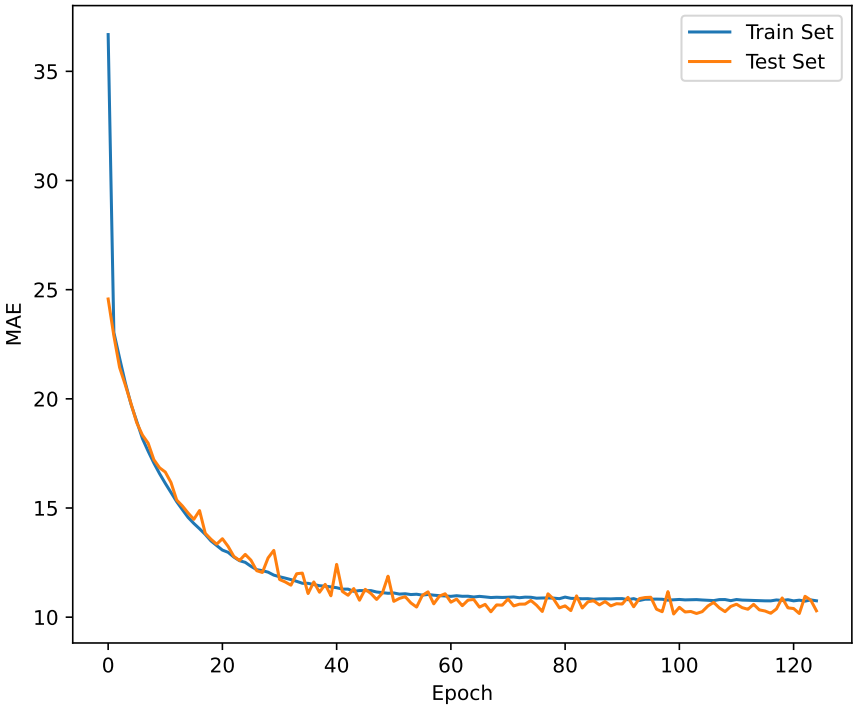


Figure 5: Example train/test curve for a neural net with a single hidden layer containing 48 neurons, trained on our example dataset (5000 training samples and 1000 testing samples) for 125 epochs.

NOTE: Similar to the linear regression model, it is occasionally possible for the neural network to produce a nonsensically high MAE and RMSE (i.e. with a MAE on the order of hundreds). If this happens, simply re-run the `NeuralNets.py` script (the neural network weights are initialised randomly, so hopefully you will get a sensible model upon re-running).

Since we specified `-save_model_to_file`, the script will have produced two files in the current directory: `NeuralNet_MODEL.json` and `NeuralNet_WEIGHTS.h5` (no `.pkl` file is produced in this case).

Now that we've trained some models, let's make some seat factor forecasts for some completely unseen data. As an example, we'll take the data in the `SEP19_FEB20.txt` file (located in the `<package_directory>\data\csvs` directory), take the September flights only, and replace the `FLOWN_CAPACITY`, `op_flown_pax` and `flown_pax` columns with dummy values (999).

To do this, create a new folder (perhaps call it `to_forecast`). Copy the

SEP19_FEB20.txt file to this folder and run the following python code:

```
import pandas as pd
import pickle
import os

df = pd.read_csv(".\\SEP19_FEB20.txt")

# Convert 'LOCAL_FLT_DT' to datetime object
df['LOCAL_FLT_DT'] = pd.to_datetime(df['LOCAL_FLT_DT'],format='%d/%m/%Y')

# Restrict to mid-late September flights only
df = df.loc[df['LOCAL_FLT_DT'] > pd.to_datetime('21/09/2019',format='%d/%m/%Y')]
df = df.loc[df['LOCAL_FLT_DT'] < pd.to_datetime('01/10/2019',format='%d/%m/%Y')]

# Reformat 'LOCAL_FLT_DT' string back to original string format
df['LOCAL_FLT_DT'] = df['LOCAL_FLT_DT'].dt.strftime("%d/%m/%Y")

# Replace columns unknown until flight day with 999s
df['FLOWN_CAPACITY'] = 999
df['op_flown_pax'] = 999
df['flown_pax'] = 999

df.to_csv(".\\SEP19_TO_FORECAST.txt",index=False)
```

This will create a `.csv` file called `SEP19_TO_FORECAST.txt`, which will act as our mock dataset for the forecasts. Of course, in a real example you would probably download this data in TeradataSQL or whatever - Just make sure that the `FLOWN_CAPACITY`, `op_flown_pax` and `flown_pax` columns are filled with dummy values like 999 (but not `NULLS`) - The values used don't matter; the code we're about to use will throw these columns away.

Now that we have a dataset on which we want to make seat factor forecasts, run the `prepare_pickles.py` script on our mock dataset:

```
python <package_directory>\utilities\prepare_pickles.py -sampling_frequency
7
```

Where we've specified `-sampling_frequency 7` as this was the sampling frequency we trained our model with (if you specify a sampling frequency that does not match that of the data used to train/test, you'll get an error later on). As usual, this will produce a `CLEANED.pkl` file. Now, run the `prepare_forecasts.py` script (found in the `<package_directory>\forecasting` folder) on the `CLEANED.pkl` file:

```
python <package_directory>\forecasting\prepare_forecasts.py CLEANED.pkl
-forecast_DTD 21
```

Where we have passed the `-forecast_DTD 21` argument since we're aiming to make predictions 21 days out. This will produce two files: `X_forecast.pkl`, which contains the data ready for input into the model, and `flt_info_for_forecast.txt`, which is a (currently partially empty) `.csv` file that will contain the seat factor forecasts.

Now, from the models we trained earlier, copy the pickled model of your choice to your current working directory (e.g. copy `LinearRegression.pkl` for linear regression or `RandomForest.pkl` for random forest or, in the case of a neural net, both `NeuralNet_MODEL.json` and `NeuralNet_WEIGHTS.h5`). As an example, to forecast using the RandomForest model, simply run:

```
python <package_directory>\forecasting\make_forecasts.py RandomForest.pkl
```

This should output:

```
Using model from file: 'RandomForest.pkl'
SUCCESS: Wrote forecasts to 'flt_info_for_forecast.txt'
```

Which tells us that the forecasting has been successful, with the predictions having been written to the `flt_info_for_forecast.txt` file. If we inspect the first few lines of this file:

```
OPG_FLT_NO,LOCAL_FLT_DT,TOTAL_CAPACITY,PREDICTED_FLOWN_PAX,PREDICTED_SF
304,2019-09-22,180,124,0.688889
304,2019-09-23,143,117,0.818182
304,2019-09-24,143,114,0.797203
304,2019-09-25,143,108,0.755245
304,2019-09-29,168,113,0.672619
```

We can see that the code has indeed predicted both the number of passengers on that flight and the corresponding seat factor. The entries in the file are ordered by `OPG_FLT_NO` and then `LOCAl_FLT_DT`.

All done! Just as a final example, we'll forecast again using the neural network as. Assuming you have already copied over both `NeuralNet_MODEL.json` and `NeuralNet_WEIGHTS.h5`, simply run:

```
python <package_directory>\forecasting\make_forecasts.py NeuralNet_MODEL.json
```

The code should again produce an output stating that the predictions have been written to the `flt_info_for_forecast.txt` file.

Things to try:

- Predicting the passengers flown at 14 days out (using information at 35,28,21,14 DTDs) and 28 days out (using information at 35,28 DTDs). Since the `sampling_frequency` is the same, we do not need to start over - But we will need to generate a new train/test set using `construct_train_test_pickles.py`.

  For 28 days out:

  ```
  python <package_directory>\utilities\construct_train_test_pickles.py
  CLEANED.pkl -train_size 5000 -test_size 1000 -forecast_DTD 28
  ```

  And for 14 days out:

  ```
  python <package_directory>\utilities\construct_train_test_pickles.py
  CLEANED.pkl -train_size 5000 -test_size 1000 -forecast_DTD 14
  ```

  Once the new train / test set has been generated, the models in `sklearn_models.py` can be run and saved to disk exactly as before.

- Using a different sampling frequency. My tests seems to indicate that weekly sampling (the default, which is also what we used in all of the examples above) is perfectly adequate and gives a reasonable trade-off between accuracy and performance.

  We can however choose to sample, say, every 3 days. This will require generating a new dataset from scratch using `prepare_pickles.py`, sampled appropriately:

  ```
  python <package_directory>\utilities\prepare_pickles.py MAR18_JUL18.txt
  AUG18.txt SEP18_DEC18 JAN19_APR19.txt -sampling_frequency 3
  ```

  We can now generate a new train/test set using `construct_train_test_pickles.py`. To make a prediction, say, 24 days out using all the information up to that point:

  ```
  python <package_directory>\utilities\construct_train_test_pickles.py
  CLEANED.pkl -train_size 5000 -test_size 1000 -forecast_DTD 24
  ```

  This may take a bit longer to run than the previous example, since due to our increased sampling, we're using more than twice as many DTDs.

Again, once finished, the files `X_train.pkl`, `y_train.pkl`, `X_test.pkl`, `y_test.pkl` should have appeared in the current directory, and modelling can be performed by running `sklearn_models.py` with the desired model name as argument.

- Experimenting with larger train/test sets. All of the code run times should remain reasonable (that is, less than an hour) for train/test sets up to 50,000 flights in size. For train/test sets of a few 100,000 flights, you're probably looking at a few hours. Again, the code has been designed so that as much information as possible is saved between steps, so if something goes wrong you'll hopefully only have to repeat the previous step.

- Trying different neural network architectures by experimenting with the `hidden_layers` argument in the `NeuralNets.py` script. The default is just a single hidden layer with 48 neurons. Passing, for example, `-hidden_layers 40 40 20` will create a neural network with three hidden layers, containg 40, 40 and 20 neurons respectively.

- Adding extra models to the `sklearn_models.py` script. This will require editing the `sklearn_models.py` file, but should be very easy as the `sklearn` API is standardized.