

Rain_in_Australia

March 30, 2024


1 Use of Machine Learning Models to Predict Odds of Rainfall in the Following Day

1.0.1 Introduction

In this project, we use data from the Kaggle dataset '[Rain in Australia](#)'. This dataset includes daily meteorological data of various cities in Australia, such as humidity, barometric pressure, wind directions etc, as well as whether it will rain the day after. This project attempts to use a Sequential Neural Network, as well as other classification models such as K-Nearest Neighbours and Decision Trees, provided by the Tensorflow-Keras and Scikit-learn library to take in the meteorological data for the day and predict whether or not it will rain tomorrow.

The data set is uploaded into Google BigQuery, and a snippet of the schema and a preview of the dataset is shown below:

 **Filter** Enter property name or value

<input type="checkbox"/>	Field name	Type	Mode	Key	Collation	Default value	Policy tags 	Description
<input type="checkbox"/>	Date	DATE	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Location	STRING	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	MinTemp	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	MaxTemp	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Rainfall	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Evaporation	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Sunshine	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	WindGustDir	STRING	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	WindGustSpeed	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	WindDir9am	STRING	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	WindDir3pm	STRING	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	WindSpeed9am	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	WindSpeed3pm	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Humidity9am	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Humidity3pm	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Pressure9am	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Pressure3pm	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Cloud9am	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Cloud3pm	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Temp9am	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	Temp3pm	FLOAT	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	RainToday	INTEGER	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	RainTomorrow	INTEGER	NULLABLE	-	-	-	-	-

Row	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am	WindDir3pm
1	2009-03-18	Cobar	13.9	29.5	0.0	6.4	11.8	ESE	28	ESE	E
2	2009-08-03	CoffsHarbour	5.7	20.2	0.0	2.0	10.0	NW	22	SW	E
3	2009-08-13	CoffsHarbour	8.3	21.2	0.0	2.6	10.2	SSW	28	W	E
4	2009-08-28	CoffsHarbour	8.4	24.0	0.0	3.0	8.6	SSW	22	WSW	E
5	2009-09-01	CoffsHarbour	4.7	20.4	0.0	3.2	10.7	SSE	30	WSW	E
6	2010-08-27	CoffsHarbour	6.3	21.6	0.0	3.0	10.8	W	33	NW	E
7	2012-05-13	CoffsHarbour	8.3	21.3	4.6	2.2	10.1	W	33	WSW	E
8	2012-08-07	CoffsHarbour	2.2	17.1	0.0	2.8	10.0	S	26	SSW	E
9	2012-08-13	CoffsHarbour	7.7	21.6	0.0	4.0	10.5	SW	26	SW	E
10	2013-09-11	CoffsHarbour	16.0	24.1	0.0	4.8	10.6	S	31	SSW	E
11	2011-01-31	Moree	19.8	36.8	0.0	11.4	14.0	NNE	33	NNE	E
12	2013-09-27	Sydney	12.7	22.6	0.0	12.2	11.4	W	46	W	E
13	2013-10-19	Sydney	12.6	22.4	0.0	6.8	12.1	ENE	37	WNW	E
14	2013-10-25	Sydney	12.2	23.4	0.0	10.8	10.4	E	35	WSW	E
15	2014-01-14	Sydney	18.3	27.5	0.0	7.2	12.0	ENE	44	W	E
16	2014-12-30	Sydney	19.4	28.1	0.0	8.0	12.7	SSW	54	S	E
17	2016-02-25	Sydney	21.9	32.8	0.0	8.4	12.2	NNE	44	NE	E
18	2017-05-25	Sydney	11.6	21.4	0.0	5.6	9.6	WSW	33	W	E
19	2009-01-16	SydneyAirport	18.4	26.3	5.4	10.2	10.1	SSE	54	S	E
20	2013-09-27	SydneyAirport	14.0	21.7	0.0	12.2	11.4	W	41	W	E
21	2016-09-05	SydneyAirport	9.4	21.3	0.0	3.4	11.1	NW	33	NW	E

1.0.2 Data Cleaning, Feature Engineering, and Exploration

Using Excel, NA and other blank values are filtered out and dropped. As some attributes e.g. wind direction or wind speed are not available at certain locations, the entire city's worth of data is dropped. However, with the vast amount of data (56420 entries in total post-cleaning), as well as

the locality of the project (as different cities have varying climates, this project only focuses on the temperate climate of Southern Australia), the dropping of data of cities in other climates will be of little effect.

After dropping of missing/non-numerical values, we turn to focus on one-hot encoding the categorical values, which appear in attributes such as wind direction. We pass the cleaned data set through the pandas library and use the `get_dummies` function:

```
[2]: import numpy as np
import pandas as pd

raw = pd.read_csv("weatherAUSclean.csv")

encoded = pd.get_dummies(raw, columns = ["WindDir9am" , "WindDir3pm"], dtype =
↳int)

#encoded.to_csv("encoded.csv")

print(encoded.head(1))
```

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	\
0	01/01/2009	Cobar	17.9	35.2	0.0	12.0	12.3	

	WindSpeed9am	WindSpeed3pm	Humidity9am	...	WindDir3pm_NW_B	\
0	6	20	20	...	0	

	WindDir3pm_N_B	WindDir3pm_SE_B	WindDir3pm_SSE_B	WindDir3pm_SSW_B	\
0	0	0	0	0	

	WindDir3pm_SW_B	WindDir3pm_S_B	WindDir3pm_WNW_B	WindDir3pm_WSW_B	\
0	1	0	0	0	

	WindDir3pm_W_B
0	0

[1 rows x 51 columns]

We can see that following the one-hot encoding, all the unique values (the 16 cardinal directions) have their dedicated column, and are assigned values 0 and 1. With the data cleaned, it is now time to split the dataset into training and testing sets. However, as this is a binary classification project, it is important our training set has an equal split of positive (Rains tomorrow) and negative (Does not rain tomorrow) data for the model to not over-train on either result and lose the ability to make predictions for the other. We use Google's BigQuery SQL to count the number of "Yes Rain" and "No Rain" for each location:

Data Skewness
▶ RUN
📄 SAVE QUERY ▾
👤 SHARE ▾
🕒 SCHEDULE
⚙️ MORE ▾

```

1 SELECT
2   Location,
3   SUM(CASE WHEN RainTomorrow = 1 THEN 1 ELSE 0 END) AS yes_rain,
4   SUM(CASE WHEN RainTomorrow = 0 THEN 1 ELSE 0 END) AS no_rain
5
6
7 FROM `loyal-rampart-414888.australia_weather.weather_data`
8
9
10 GROUP BY Location
11

```

The data is saved as a .csv file and imported into Excel to calculate the percentage of “Yes Rain”:

	A	B	C	D
1	Location	yes_rain	no_rain	yes_percentage
2	Portland	746	1117	0.400429415
3	CoffsHarbour	431	949	0.312318841
4	Cairns	751	1693	0.307283142
5	NorfolkIsland	743	1721	0.301542208
6	MountGambier	732	1733	0.296957404
7	Williamtown	320	878	0.267111853
8	Darwin	789	2273	0.257674722
9	Watsonia	684	2046	0.250549451
10	SydneyAirport	715	2155	0.24912892
11	Melbourne	471	1427	0.248155954
12	Sydney	416	1274	0.246153846
13	Hobart	468	1471	0.241361527
14	Brisbane	642	2311	0.217406028
15	MelbourneAirport	636	2293	0.217138955
16	Sale	359	1319	0.213945173
17	Perth	616	2409	0.203636364
18	Canberra	219	859	0.203153989
19	Nuriootpa	389	1619	0.1937251
20	PerthAirport	553	2360	0.189838654
21	WaggaWagga	427	1989	0.176738411
22	Townsville	394	2025	0.162877222
23	Moree	258	1655	0.134866702
24	Cobar	63	471	0.117977528
25	Mildura	289	2305	0.111410948
26	AliceSprings	187	2036	0.084120558
27	Woomera	129	1605	0.074394464

Plotting this information on as a histogram,

```
[3]: import seaborn as sns
import matplotlib.pyplot as plt

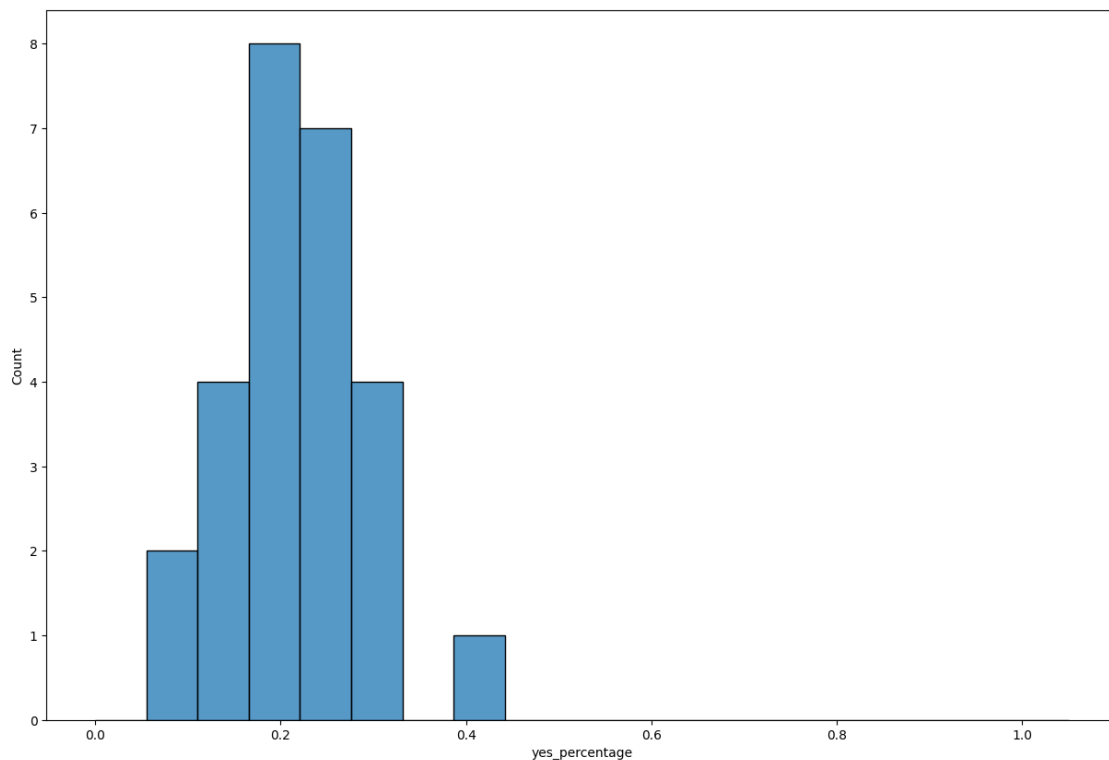
raw = pd.read_csv(r"label_skew.csv")

n = 20

fig, ax = plt.subplots(figsize = (15, 10))

sns.histplot(ax = ax, data = raw, x = "yes_percentage", bins = np.linspace(0, 1,
↪ + 1/n, n))

plt.savefig(r"figs\data_skew.png")
```



We can see that much of the data is skewed towards the left, meaning that much of the data is deficient in “Yes Rain” instances, and so we have to be mindful when splitting our training data.

From the .csv file, we see that the location with the largest portion of “Yes Rain” data is Portland, which resides in a temperate climate. Melbourne, another city in the temperate climate of South Australia, while doesn’t have a particularly high “Yes Rain” ratio, the number of data entries of the “Yes Rain” type is numerous, so it would be good to include data from Melbourne as well for our training set. The final pool of training locations include:

- Portland

- Melbourne Airport
- Melbourne
- Watsonia
- Mount Gambier

1.0.3 Data Splitting and Normalisation

In this section, we create a python function which allows us to split the dataframe into the training and testing set. We need to create a custom function which selects a portion of the “Yes Rain” data, and select an equal number of “No Rain” data to ensure fair training opportunities of the model:

```
[4]: def DataSplitter(dataframe, n_yes, want_test):

    if n_yes == None:
        n_yes = int(len(dataframe[dataframe["RainTomorrow"] == 1]) * 0.9)
    else:
        pass

    yes_data = dataframe[dataframe["RainTomorrow"] == 1].sample(n = n_yes)
    no_data = dataframe[dataframe["RainTomorrow"] == 0].sample(n = n_yes)

    try:
        splitted_pool = pd.concat([yes_data, no_data], axis = 0).drop(columns = 'Unnamed: 0')
    except KeyError:
        splitted_pool = pd.concat([yes_data, no_data], axis = 0)

    if want_test == True:
        try:
            test_pool = (dataframe.drop(labels = list(splitted_pool.index.
values))).drop(columns = ['Unnamed: 0'])
        except KeyError:
            test_pool = dataframe.drop(labels = list(splitted_pool.index.
values))

    return [splitted_pool, test_pool]
else:
    return splitted_pool
```

In the above function, three parameters are taken in.

1. **dataframe**: The `dataframe` parameter takes in the dataframe needed to be split.
2. **n_yes**: The `n_yes` parameter tells the function the number of “Yes Rain” and subsequently “No Rain” data used to form the training set. If no integer is passed to the argument, then it automatically takes 90% of the “Yes Rain” data and uses it as training data.
3. **want_test**: If set to `True`, the function returns the remainder of the `dataframe` not used as training data as test data.

With the data splitting function defined, we import the one-hot encoded, clean data .csv as a pandas dataframe:

```
[5]: data = pd.read_csv("encoded.csv")

loc = ["Portland", "MelbourneAirport", "Melbourne", "Watsonia", "Mount Gambier"]

location_data = data[data["Location"].isin(loc)]

train_pool, test_pool = DataSplitter(location_data, n_yes = None, want_test = True) #train pool now has equal split of yes/no rain

# Number of training data with "Yes Rain" / "No Rain"
print("Yes Rain Size:", len(train_pool[train_pool["RainTomorrow"] == 1]))
print("No Rain Size:", len(train_pool[train_pool["RainTomorrow"] == 0]))

feature_labels = (train_pool.drop(columns = ["Date", "Location", "RainTomorrow"])).columns

X_train = train_pool.drop(columns = ["Date", "Location", "RainTomorrow"])

Y_train = train_pool["RainTomorrow"]
```

Yes Rain Size: 2283

No Rain Size: 2283

We can see that now the training set has an equal balance of “Yes Rain” and “No Rain” data. The DataSplitter function has also returned the test dataframe, which will be further transformed later on.

```
[5]: X_train.describe()
```

```
[5]:
```

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	\
count	4566.000000	4566.000000	4566.000000	4566.000000	4566.000000	
mean	10.308782	19.298182	2.497985	4.241524	5.585983	
std	4.299058	6.242181	5.915478	3.204399	3.769308	
min	-1.200000	8.400000	0.000000	0.000000	0.000000	
25%	7.300000	14.600000	0.000000	2.000000	2.400000	
50%	9.800000	17.600000	0.200000	3.400000	5.400000	
75%	13.100000	22.600000	2.200000	5.800000	8.500000	
max	28.800000	46.800000	84.000000	23.800000	13.800000	

	WindSpeed9am	WindSpeed3pm	Humidity9am	Humidity3pm	Pressure9am	...	\
count	4566.000000	4566.000000	4566.000000	4566.000000	4566.000000	...	
mean	17.591546	21.154402	74.395532	59.068988	1016.389159	...	
std	10.555805	9.711638	16.513473	19.225242	8.153892	...	
min	2.000000	2.000000	11.000000	6.000000	988.900000	...	
25%	9.000000	15.000000	64.000000	46.000000	1011.000000	...	

50%	15.000000	20.000000	75.000000	59.000000	1016.700000	...
75%	24.000000	28.000000	87.000000	72.000000	1022.000000	...
max	65.000000	76.000000	100.000000	100.000000	1038.200000	...

	SSW_B	SW_B	S_B	WNW_B	WSW_B	\
count	4566.000000	4566.000000	4566.000000	4566.000000	4566.000000	
mean	0.105563	0.090670	0.125055	0.049715	0.082348	
std	0.307311	0.287171	0.330817	0.217380	0.274924	
min	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	0.000000	0.000000	0.000000	0.000000	
50%	0.000000	0.000000	0.000000	0.000000	0.000000	
75%	0.000000	0.000000	0.000000	0.000000	0.000000	
max	1.000000	1.000000	1.000000	1.000000	1.000000	

	W_B	Spring	Summer	Autumn	Winter
count	4566.000000	4566.000000	4566.000000	4566.000000	4566.000000
mean	0.080815	0.259089	0.199956	0.253176	0.287779
std	0.272580	0.438182	0.400011	0.434878	0.452777
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	1.000000	0.000000	1.000000	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000

[8 rows x 52 columns]

Using the `.describe()` method on the training set dataframe, we can see that the min/max and statistical structure of each feature is vastly different. In order to make training the model more efficient and easier, we must normalise all these columns. We use the `StandardScaler` function provided by scikit-learn. Once again, as the normalisation process will have to be applied later on again, we define a function which makes calling the normalising process easier:

```
[6]: from sklearn.preprocessing import StandardScaler
from sklearn.compose import make_column_transformer

def DataNormaliser(dataframe, normaliser):

    if normaliser == None:
        normaliser = make_column_transformer((
            StandardScaler(), ['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation',
↪ 'Sunshine',
                                'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am',
↪ 'Humidity3pm', 'Pressure9am', 'Pressure3pm', 'Cloud9am', 'Cloud3pm',
↪ 'Temp9am', 'Temp3pm']
        ))

    normaliser.fit(dataframe)
```



```

else:
    pass

    dataframe_norm = pd.DataFrame(data = normaliser.transform(dataframe))
    dataframe_onehot = dataframe.iloc[:, 15:]

    normalised_df = pd.concat([dataframe_norm.reset_index(drop = True),
↪dataframe_onehot.reset_index(drop = True)], axis = 1)
    normalised_df.columns = dataframe.columns

    return [normaliser, normalised_df]

```

In the `DataNormaliser` function, two arguments must be passed:

1. **dataframe**: the dataframe to be normalised
2. **normaliser**: the normaliser to apply. If `None` is passed to `normaliser`, a new normaliser will be produced and fitted onto the passed `dataframe`. The new fitted normaliser object will then be returned along with the normalised dataframe

We create a normaliser object `normaliser` with the normalising method `StandardScaler()`, normalising the dataset using the z-value method. We pass the names of the features of the dataframe which needs normalising. Note that all one-hot encoded (categorical) features are not passed into the list, as they are already normaliser (only 0, 1 values). The normaliser is then fit to the dataframe (limited to the mentioned features). It is then concatenated with the remaining one-hot encoded features to reform the full normalised training data set.

Using the function to normalise the training data `X_train`:

```

[7]: adapted_normaliser, X_train_norm = DataNormaliser(X_train, normaliser = None)

X_train_norm.describe()

```

```

[7]:
      MinTemp      MaxTemp      Rainfall      Evaporation      Sunshine \
count  4.566000e+03  4.566000e+03  4.566000e+03  4.566000e+03  4.566000e+03
mean   -2.396487e-16  5.975655e-16 -2.489856e-17  4.357248e-17  4.979713e-17
std     1.000110e+00  1.000110e+00  1.000110e+00  1.000110e+00  1.000110e+00
min    -2.894508e+00 -1.747318e+00 -4.099871e-01 -1.307121e+00 -1.472932e+00
25%    -7.109210e-01 -7.670409e-01 -4.099871e-01 -7.567501e-01 -8.425755e-01
50%    -1.301797e-01 -2.610914e-01 -3.761837e-01 -2.675314e-01 -5.462933e-02
75%     6.363989e-01  5.452655e-01 -3.814918e-02  4.662966e-01  7.858466e-01
max     4.678359e+00  4.324076e+00  1.378746e+01  5.970006e+00  2.177885e+00

      WindSpeed9am  WindSpeed3pm  Humidity9am  Humidity3pm  Pressure9am \
count  4.566000e+03  4.566000e+03  4.566000e+03  4.566000e+03  4.566000e+03
mean   -1.493914e-16  1.742899e-16  2.987828e-16  4.979713e-17  3.709886e-15
std     1.000110e+00  1.000110e+00  1.000110e+00  1.000110e+00  1.000110e+00
min    -1.486259e+00 -1.977848e+00 -3.800134e+00 -2.689815e+00 -3.327743e+00
25%    -8.143901e-01 -8.288446e-01 -6.159808e-01 -6.710509e-01 -6.533114e-01
50%    -2.385026e-01 -9.766050e-02  4.488114e-02  1.870549e-03  5.179437e-02

```

75%	5.773378e-01	5.290687e-01	7.658215e-01	6.747920e-01	6.854577e-01
max	4.560559e+00	4.602809e+00	1.546840e+00	2.124161e+00	2.859793e+00

	...	SSW_B	SW_B	S_B	WNW_B	WSW_B \
count	...	4566.000000	4566.000000	4566.000000	4566.000000	4566.000000
mean	...	0.107753	0.088042	0.125712	0.047744	0.080596
std	...	0.310102	0.283387	0.331561	0.213248	0.272243
min	...	0.000000	0.000000	0.000000	0.000000	0.000000
25%	...	0.000000	0.000000	0.000000	0.000000	0.000000
50%	...	0.000000	0.000000	0.000000	0.000000	0.000000
75%	...	0.000000	0.000000	0.000000	0.000000	0.000000
max	...	1.000000	1.000000	1.000000	1.000000	1.000000

	W_B	Spring	Summer	Autumn	Winter
count	4566.000000	4566.000000	4566.000000	4566.000000	4566.000000
mean	0.081472	0.257556	0.202584	0.252738	0.287122
std	0.273588	0.437336	0.401969	0.434629	0.452469
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	1.000000	0.000000	1.000000	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000

[8 rows x 52 columns]

We can see that post-normalising, the training data is much more statistically consistent, with a mean of 0 and a standard deviation of 1 as expected.

Now moving to the testing dataset, we split it into the feature frame and label frame as usual. We then normalise the feature frame using the already-fitted normaliser `adapted_normaliser` to be consistent with the training frame in order for model to be evaluated properly.

```
[8]: X_test = test_pool.drop(columns = ["Date", "Location", "RainTomorrow"])
      Y_test = test_pool["RainTomorrow"]

      X_test_norm = DataNormaliser(X_test, adapted_normaliser)[1]

      X_test_norm.describe()
```

[8]:	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine \
count	4854.000000	4854.000000	4854.000000	4854.000000	4854.000000
mean	-0.021124	0.174696	-0.160904	0.065594	0.376117
std	0.996587	1.000535	0.729594	0.977469	0.992921
min	-2.755131	-1.652452	-0.409987	-1.307121	-1.472932
25%	-0.710921	-0.608932	-0.409987	-0.695598	-0.396073
50%	-0.060491	-0.023928	-0.409987	-0.145227	0.470668
75%	0.682858	0.734997	-0.274773	0.588601	1.179820
max	4.120847	4.308265	13.618444	5.664245	2.282944

	WindSpeed9am	WindSpeed3pm	Humidity9am	Humidity3pm	Pressure9am	...	\
count	4854.000000	4854.000000	4854.000000	4854.000000	4854.000000	...	
mean	-0.149403	-0.112035	-0.098854	-0.369504	0.360983	...	
std	0.976762	0.946093	0.965172	0.842601	0.905998	...	
min	-1.486259	-1.977848	-3.800134	-2.741579	-3.402292	...	
25%	-0.814390	-0.828845	-0.736138	-0.929867	-0.230869	...	
50%	-0.430465	-0.202115	-0.075276	-0.360472	0.387263	...	
75%	0.241404	0.529069	0.585586	0.208923	0.958803	...	
max	4.752522	5.751812	1.546840	2.124161	2.847368	...	

	SSW_B	SW_B	S_B	WNW_B	WSW_B	...	\
count	4854.000000	4854.000000	4854.000000	4854.000000	4854.000000	...	
mean	0.108570	0.086527	0.165019	0.028018	0.066543	...	
std	0.311131	0.281169	0.371236	0.165042	0.249255	...	
min	0.000000	0.000000	0.000000	0.000000	0.000000	...	
25%	0.000000	0.000000	0.000000	0.000000	0.000000	...	
50%	0.000000	0.000000	0.000000	0.000000	0.000000	...	
75%	0.000000	0.000000	0.000000	0.000000	0.000000	...	
max	1.000000	1.000000	1.000000	1.000000	1.000000	...	

	W_B	Spring	Summer	Autumn	Winter	...	\
count	4854.000000	4854.000000	4854.000000	4854.000000	4854.000000	...	
mean	0.05789	0.242068	0.264730	0.261640	0.231562	...	
std	0.23356	0.428380	0.441235	0.439573	0.421874	...	
min	0.000000	0.000000	0.000000	0.000000	0.000000	...	
25%	0.000000	0.000000	0.000000	0.000000	0.000000	...	
50%	0.000000	0.000000	0.000000	0.000000	0.000000	...	
75%	0.000000	0.000000	1.000000	1.000000	0.000000	...	
max	1.000000	1.000000	1.000000	1.000000	1.000000	...	

[8 rows x 52 columns]

As the test dataframe is normalised using the normaliser fitted with the training data, it is expected that the mean and standard deviations are not 0 and 1 respectively. However, the variation in the data is reduced compared to the un-normalised version, which makes it ready for evaluation later on.

1.0.4 Model Building and Training

We use a Sequential neural network provided by the keras library. We begin with a simple model of two hidden layers (excluding the output layer), each layer having 16 nodes. The activation function used for the hidden layers is the standard ReLU to account for non-linear relationships in the 50+ features in the training set. However, as we are dealing with a binary classification problem, the activation function for the output layer is set to be a sigmoid curve to constrain possible outputs in the range of 0 to 1. Binary Crossentropy is used as the loss function, and the keras optimiser Adam is used to compile the model. We choose Binary Accuracy as the metric to evaluate the model, and set the classification threshold to be 0.7.

We fit the model using an initial learning rate of 0.01 for 50 epochs, as well as a validation set, which is 20% of the training data.

```
[10]: from tensorflow import keras
model = keras.models.Sequential(
    [
        keras.layers.Dense(units = 16, activation = "relu"),
        keras.layers.Dense(units = 16, activation = "relu"),
        keras.layers.Dense(units = 1, activation = "sigmoid")
    ]
)

metric = keras.metrics.BinaryAccuracy(threshold=0.7)
model.compile(loss = "binary_crossentropy", optimizer = keras.optimizers.
    Adam(learning_rate = 0.01), metrics = metric)

history = model.fit(X_train_norm, Y_train, epochs = 50, verbose = 1,
    validation_split = 0.2)
```

WARNING:tensorflow:From C:\Users\User\anaconda3\envs\myenv\Lib\site-packages\keras\src\backend.py:873: The name tf.get_default_graph is deprecated. Please use tf.compat.v1.get_default_graph instead.

Epoch 1/50

WARNING:tensorflow:From C:\Users\User\anaconda3\envs\myenv\Lib\site-packages\keras\src\utils\tf_utils.py:492: The name tf.ragged.RaggedTensorValue is deprecated. Please use tf.compat.v1.ragged.RaggedTensorValue instead.

```
115/115 [=====] - 2s 5ms/step - loss: 0.4828 -
binary_accuracy: 0.7185 - val_loss: 0.7099 - val_binary_accuracy: 0.7746
Epoch 2/50
115/115 [=====] - 0s 3ms/step - loss: 0.4365 -
binary_accuracy: 0.7525 - val_loss: 0.5545 - val_binary_accuracy: 0.8315
Epoch 3/50
115/115 [=====] - 0s 3ms/step - loss: 0.4224 -
binary_accuracy: 0.7700 - val_loss: 0.6716 - val_binary_accuracy: 0.8293
Epoch 4/50
115/115 [=====] - 0s 3ms/step - loss: 0.4038 -
binary_accuracy: 0.7807 - val_loss: 0.6250 - val_binary_accuracy: 0.8326
Epoch 5/50
115/115 [=====] - 0s 3ms/step - loss: 0.4028 -
binary_accuracy: 0.7788 - val_loss: 0.7802 - val_binary_accuracy: 0.7812
Epoch 6/50
115/115 [=====] - 0s 3ms/step - loss: 0.3903 -
binary_accuracy: 0.7903 - val_loss: 0.6667 - val_binary_accuracy: 0.8096
Epoch 7/50
115/115 [=====] - 0s 3ms/step - loss: 0.3842 -
binary_accuracy: 0.7965 - val_loss: 0.6018 - val_binary_accuracy: 0.8457
```

Epoch 8/50
115/115 [=====] - 0s 4ms/step - loss: 0.3715 -
binary_accuracy: 0.7998 - val_loss: 0.7599 - val_binary_accuracy: 0.7877
Epoch 9/50
115/115 [=====] - 0s 3ms/step - loss: 0.3652 -
binary_accuracy: 0.8089 - val_loss: 0.6104 - val_binary_accuracy: 0.8435
Epoch 10/50
115/115 [=====] - 0s 3ms/step - loss: 0.3644 -
binary_accuracy: 0.8048 - val_loss: 0.6513 - val_binary_accuracy: 0.8337
Epoch 11/50
115/115 [=====] - 0s 3ms/step - loss: 0.3583 -
binary_accuracy: 0.8039 - val_loss: 0.5782 - val_binary_accuracy: 0.8457
Epoch 12/50
115/115 [=====] - 0s 3ms/step - loss: 0.3441 -
binary_accuracy: 0.8163 - val_loss: 0.6464 - val_binary_accuracy: 0.8249
Epoch 13/50
115/115 [=====] - 0s 3ms/step - loss: 0.3342 -
binary_accuracy: 0.8245 - val_loss: 0.6715 - val_binary_accuracy: 0.8118
Epoch 14/50
115/115 [=====] - 0s 3ms/step - loss: 0.3275 -
binary_accuracy: 0.8228 - val_loss: 0.6176 - val_binary_accuracy: 0.8239
Epoch 15/50
115/115 [=====] - 0s 4ms/step - loss: 0.3235 -
binary_accuracy: 0.8321 - val_loss: 0.6658 - val_binary_accuracy: 0.8260
Epoch 16/50
115/115 [=====] - 0s 3ms/step - loss: 0.3180 -
binary_accuracy: 0.8354 - val_loss: 0.7431 - val_binary_accuracy: 0.7856
Epoch 17/50
115/115 [=====] - 0s 3ms/step - loss: 0.3177 -
binary_accuracy: 0.8379 - val_loss: 0.6711 - val_binary_accuracy: 0.8217
Epoch 18/50
115/115 [=====] - 0s 3ms/step - loss: 0.3021 -
binary_accuracy: 0.8436 - val_loss: 0.6774 - val_binary_accuracy: 0.8107
Epoch 19/50
115/115 [=====] - 0s 3ms/step - loss: 0.2985 -
binary_accuracy: 0.8505 - val_loss: 0.7536 - val_binary_accuracy: 0.7932
Epoch 20/50
115/115 [=====] - 0s 3ms/step - loss: 0.2915 -
binary_accuracy: 0.8521 - val_loss: 0.8653 - val_binary_accuracy: 0.7779
Epoch 21/50
115/115 [=====] - 0s 3ms/step - loss: 0.2900 -
binary_accuracy: 0.8486 - val_loss: 0.9214 - val_binary_accuracy: 0.7505
Epoch 22/50
115/115 [=====] - 0s 4ms/step - loss: 0.2832 -
binary_accuracy: 0.8584 - val_loss: 1.0671 - val_binary_accuracy: 0.7177
Epoch 23/50
115/115 [=====] - 0s 3ms/step - loss: 0.2781 -
binary_accuracy: 0.8598 - val_loss: 0.8330 - val_binary_accuracy: 0.7790

Epoch 24/50
115/115 [=====] - 0s 3ms/step - loss: 0.2750 -
binary_accuracy: 0.8634 - val_loss: 0.6897 - val_binary_accuracy: 0.8151
Epoch 25/50
115/115 [=====] - 0s 3ms/step - loss: 0.2647 -
binary_accuracy: 0.8634 - val_loss: 0.9020 - val_binary_accuracy: 0.7659
Epoch 26/50
115/115 [=====] - 0s 3ms/step - loss: 0.2635 -
binary_accuracy: 0.8710 - val_loss: 0.8913 - val_binary_accuracy: 0.7691
Epoch 27/50
115/115 [=====] - 0s 3ms/step - loss: 0.2600 -
binary_accuracy: 0.8710 - val_loss: 0.6889 - val_binary_accuracy: 0.8228
Epoch 28/50
115/115 [=====] - 0s 3ms/step - loss: 0.2565 -
binary_accuracy: 0.8716 - val_loss: 0.9856 - val_binary_accuracy: 0.7505
Epoch 29/50
115/115 [=====] - 0s 4ms/step - loss: 0.2498 -
binary_accuracy: 0.8749 - val_loss: 0.9695 - val_binary_accuracy: 0.7604
Epoch 30/50
115/115 [=====] - 0s 3ms/step - loss: 0.2508 -
binary_accuracy: 0.8781 - val_loss: 0.8810 - val_binary_accuracy: 0.7713
Epoch 31/50
115/115 [=====] - 0s 3ms/step - loss: 0.2467 -
binary_accuracy: 0.8787 - val_loss: 1.1105 - val_binary_accuracy: 0.7287
Epoch 32/50
115/115 [=====] - 0s 3ms/step - loss: 0.2424 -
binary_accuracy: 0.8779 - val_loss: 1.0812 - val_binary_accuracy: 0.7473
Epoch 33/50
115/115 [=====] - 0s 3ms/step - loss: 0.2396 -
binary_accuracy: 0.8771 - val_loss: 1.1968 - val_binary_accuracy: 0.7243
Epoch 34/50
115/115 [=====] - 0s 3ms/step - loss: 0.2438 -
binary_accuracy: 0.8787 - val_loss: 1.2783 - val_binary_accuracy: 0.7123
Epoch 35/50
115/115 [=====] - 0s 3ms/step - loss: 0.2385 -
binary_accuracy: 0.8823 - val_loss: 0.9444 - val_binary_accuracy: 0.7757
Epoch 36/50
115/115 [=====] - 0s 3ms/step - loss: 0.2270 -
binary_accuracy: 0.8853 - val_loss: 1.0618 - val_binary_accuracy: 0.7440
Epoch 37/50
115/115 [=====] - 0s 3ms/step - loss: 0.2306 -
binary_accuracy: 0.8839 - val_loss: 1.0679 - val_binary_accuracy: 0.7484
Epoch 38/50
115/115 [=====] - 0s 4ms/step - loss: 0.2238 -
binary_accuracy: 0.8880 - val_loss: 0.9250 - val_binary_accuracy: 0.7867
Epoch 39/50
115/115 [=====] - 0s 3ms/step - loss: 0.2339 -
binary_accuracy: 0.8781 - val_loss: 1.0538 - val_binary_accuracy: 0.7615

```

Epoch 40/50
115/115 [=====] - 0s 4ms/step - loss: 0.2322 -
binary_accuracy: 0.8872 - val_loss: 1.1309 - val_binary_accuracy: 0.7549
Epoch 41/50
115/115 [=====] - 0s 3ms/step - loss: 0.2264 -
binary_accuracy: 0.8825 - val_loss: 1.0705 - val_binary_accuracy: 0.7593
Epoch 42/50
115/115 [=====] - 0s 3ms/step - loss: 0.2166 -
binary_accuracy: 0.8954 - val_loss: 1.0618 - val_binary_accuracy: 0.7702
Epoch 43/50
115/115 [=====] - 0s 3ms/step - loss: 0.2208 -
binary_accuracy: 0.8932 - val_loss: 1.1573 - val_binary_accuracy: 0.7538
Epoch 44/50
115/115 [=====] - 0s 3ms/step - loss: 0.2167 -
binary_accuracy: 0.8943 - val_loss: 1.1430 - val_binary_accuracy: 0.7495
Epoch 45/50
115/115 [=====] - 0s 3ms/step - loss: 0.2095 -
binary_accuracy: 0.8954 - val_loss: 1.4544 - val_binary_accuracy: 0.7112
Epoch 46/50
115/115 [=====] - 0s 3ms/step - loss: 0.2179 -
binary_accuracy: 0.8973 - val_loss: 1.1239 - val_binary_accuracy: 0.7473
Epoch 47/50
115/115 [=====] - 0s 3ms/step - loss: 0.2091 -
binary_accuracy: 0.9012 - val_loss: 1.0662 - val_binary_accuracy: 0.7604
Epoch 48/50
115/115 [=====] - 0s 3ms/step - loss: 0.2010 -
binary_accuracy: 0.9031 - val_loss: 1.4175 - val_binary_accuracy: 0.7123
Epoch 49/50
115/115 [=====] - 0s 3ms/step - loss: 0.1965 -
binary_accuracy: 0.9080 - val_loss: 1.2814 - val_binary_accuracy: 0.7440
Epoch 50/50
115/115 [=====] - 0s 3ms/step - loss: 0.2037 -
binary_accuracy: 0.9017 - val_loss: 1.2346 - val_binary_accuracy: 0.7484

```

Evaluating the model using the test set,

```
[11]: model.evaluate(X_test_norm, Y_test)
```

```

152/152 [=====] - 0s 2ms/step - loss: 1.1488 -
binary_accuracy: 0.7604

```

```
[11]: [1.1487643718719482, 0.7604038119316101]
```

We get a log loss of 1.149 and an accuracy of 0.76. We can pass the information from the `history` object to a pyplot plot and plot the evolution of the binary accuracy metric, loss, validation loss as the epoch increases. We can further probe the model and find relative weighting/importance each training feature has on the final prediction, using the `tf.GradientTape()` method.

```
[13]: import tensorflow as tf

#plotting model metrics and losses
figL, axL = plt.subplots(figsize = (14, 12))

axL.plot(history.history["loss"], color = "blue", lw = 1, alpha = 0.6)
axL.plot(history.history["val_loss"], color = "orange", lw = 1, alpha = 0.6)
axL.plot(history.history["binary_accuracy"], color = "green", lw = 1, alpha = 0.6)

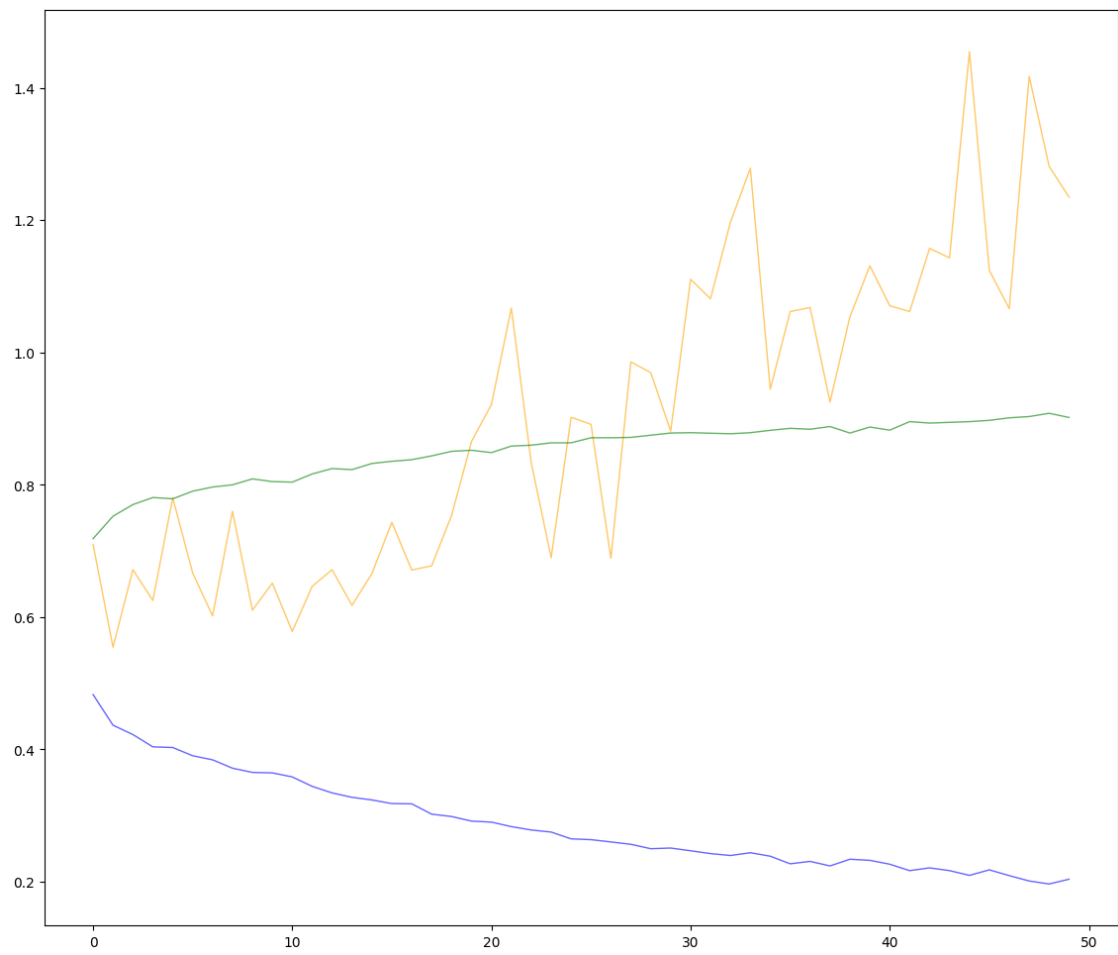
#plotting feature importance
input_tensor = tf.convert_to_tensor(X_train_norm)

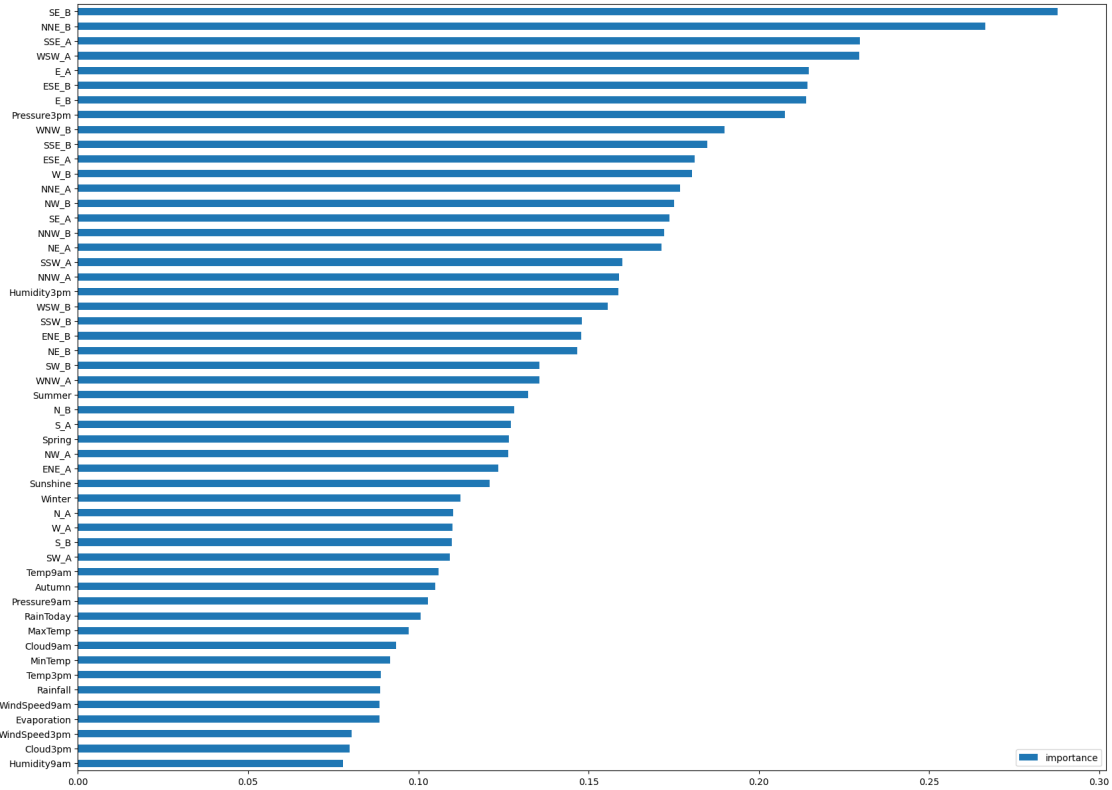
with tf.GradientTape() as tape:
    tape.watch(input_tensor)
    output = model(input_tensor)

gradients = tape.gradient(output, input_tensor)
feat_importance = np.mean(np.abs(gradients.numpy()), axis = 0)

figF, axF = plt.subplots(figsize = (20, 15))
feat_importance_arr = np.reshape(feat_importance, (len(feat_importance), 1))
feat_importance_df = pd.DataFrame(data = feat_importance_arr, columns = ["importance"], index = feature_labels).sort_values(by = ["importance"], ascending = True)
feat_importance_df.plot(kind = 'barh', ax = axF)
```

```
[13]: <Axes: >
```



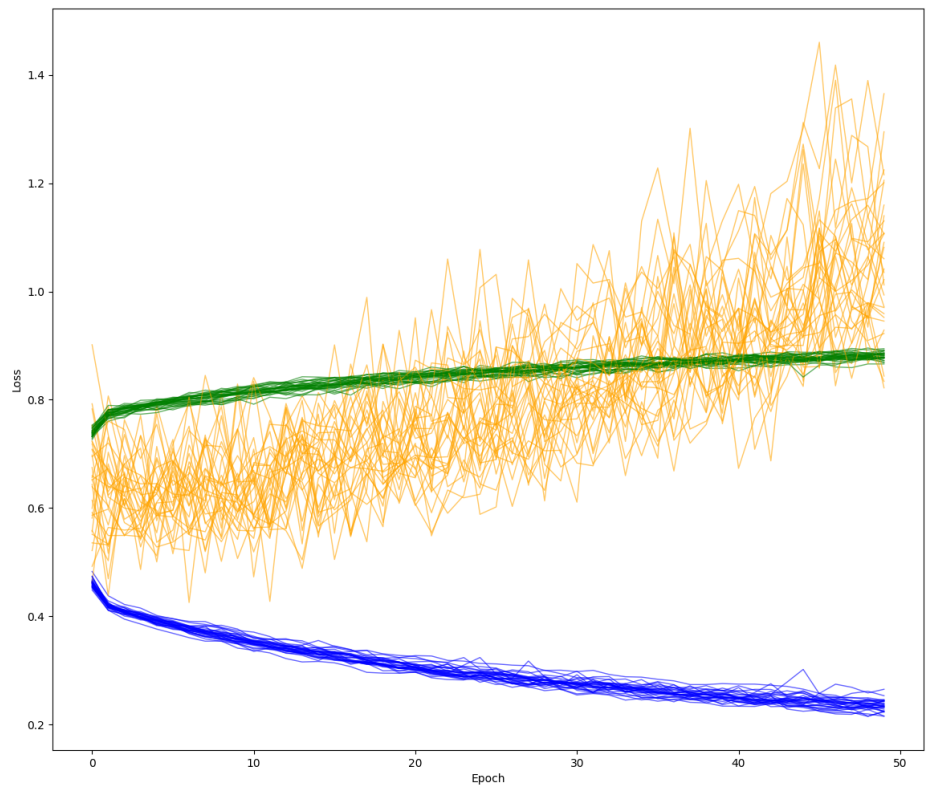
1.0.5 Model Evaluation

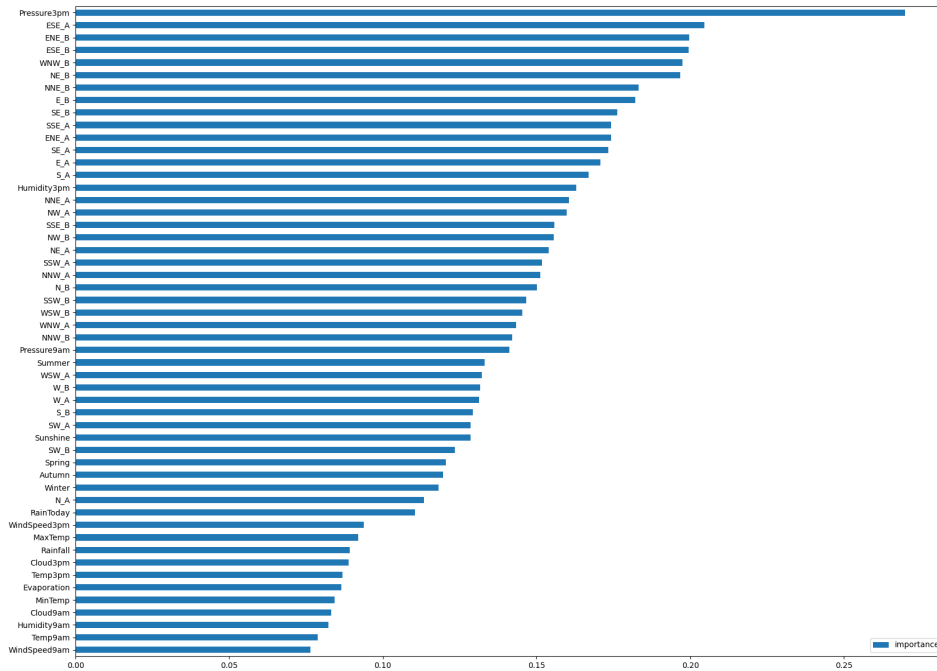
From the graphs above, we can make two observations:

1. The validation loss (orange) diverges from the training loss (blue).
2. Many features contribute towards the final prediction

These two observations together suggest that it is possible that the model is overfitting the data. As a result, although the training loss is decreasing, the validation loss is increasing with epoch number, suggesting that the model is training too specifically and picking up all the noises and not generalising enough. Point 2 is also evidence for this, as all the different wind directions seem to play a part, which does not seem to be a realistic conclusion.

However, as the model was tested on a random sample of the full dataset (the `.sample()` method is used in the `DataSplitter` function), it is better to evaluate the model multiple times on random samples of the full dataset to gain an averaged evaluation of the model before deciding on how to tweak the hyperparameters to increase the accuracy of the model and reduce its loss. We run a K-Fold cross-validation for k=30 times, and plot its loss-epoch function and its feature importance again.





Once again, through these graphs, the 2 issues of diverging validation loss and odd feature importance are persistent, even after running a k=30 fold cross-validation. Hence, we can introduce some methods of bringing this loss-divergence down, and hone the model to more decisively pick out the key features. Here, we show the aggregate data for each fold's loss and accuracy evaluation:

```
[14]: aggregate = pd.read_csv(r"model data\base\model_eval.txt")
print(aggregate, "\n")
print("Mean loss:", aggregate["loss"].mean())
print("Mean accuracy:", aggregate["binary_accuracy"].mean())
```

	loss	binary_accuracy
0	0.884622	0.750208
1	1.282568	0.705882
2	1.083097	0.759373
3	0.982834	0.762706
4	0.964681	0.743376
5	0.993536	0.761206
6	1.198444	0.727379
7	1.146771	0.727045
8	0.918404	0.773704
9	1.138888	0.754708
10	0.979593	0.758040
11	1.071732	0.767039
12	1.051185	0.758707

13	1.059582	0.756541
14	0.902933	0.794368
15	0.818326	0.779370
16	1.168685	0.742210
17	0.903191	0.762540
18	0.949083	0.771538
19	1.375025	0.737377
20	0.965554	0.758540
21	0.883518	0.750042
22	1.142419	0.766206
23	1.054085	0.772538
24	1.083135	0.738377
25	0.892387	0.769872
26	1.001720	0.748542
27	1.115479	0.753874
28	1.047207	0.728212
29	1.113738	0.750208

Mean loss: 1.0390807072321573

Mean accuracy: 0.7543242792288463

Next, we can try to improve this model. We can see that while training loss by epoch is low, validation loss by epoch is high, which suggests overfitting. First, we can reduce the learning rate to ensure that the training/validation losses are more stable instead of jittering as seen in the above figures. We select a small learning rate of 0.0002.

Secondly, to prevent the domination of random features leading to overfitting of the training data, we introduce an L1 regularising scheme for the two hidden layers to encourage irrelevant or weak features to decrease. We use a regularisation parameter of 0.05 to make sure the regularising scheme is not so strong as to completely kill all feature effects. Compiling the new model and subjecting it to a k=30 fold cross validation again, we get

```
model = keras.models.Sequential(
    [
        keras.layers.Dense(units = 16, activation = "relu", kernel_regularizer=keras.regularizers.l1(0.05)),
        keras.layers.Dense(units = 16, activation = "relu", kernel_regularizer=keras.regularizers.l1(0.05)),
        keras.layers.Dense(units = 1, activation = "sigmoid")
    ]
)
```

```
thresh = 0.7
```

```
metrics = keras.metrics.BinaryAccuracy(threshold=thresh)
```

```
max_epoch, min_lr, max_lr, n = 100, 0.0002, 0.01, 4
```

```
def AdaptiveLearning(epoch):
    if epoch == 0:
        return max_lr
```

```

else:
    return (max_lr-min_lr)/(max_epoch**n)*(epoch-max_epoch)**n+min_lr

lr_scheduler = keras.callbacks.LearningRateScheduler(AdaptiveLearning)

model.compile(loss = "binary_crossentropy", optimizer = keras.optimizers.Adam(), metrics = met

history = model.fit(X_train_norm, Y_train, epochs = max_epoch, verbose = 1, validation_split =

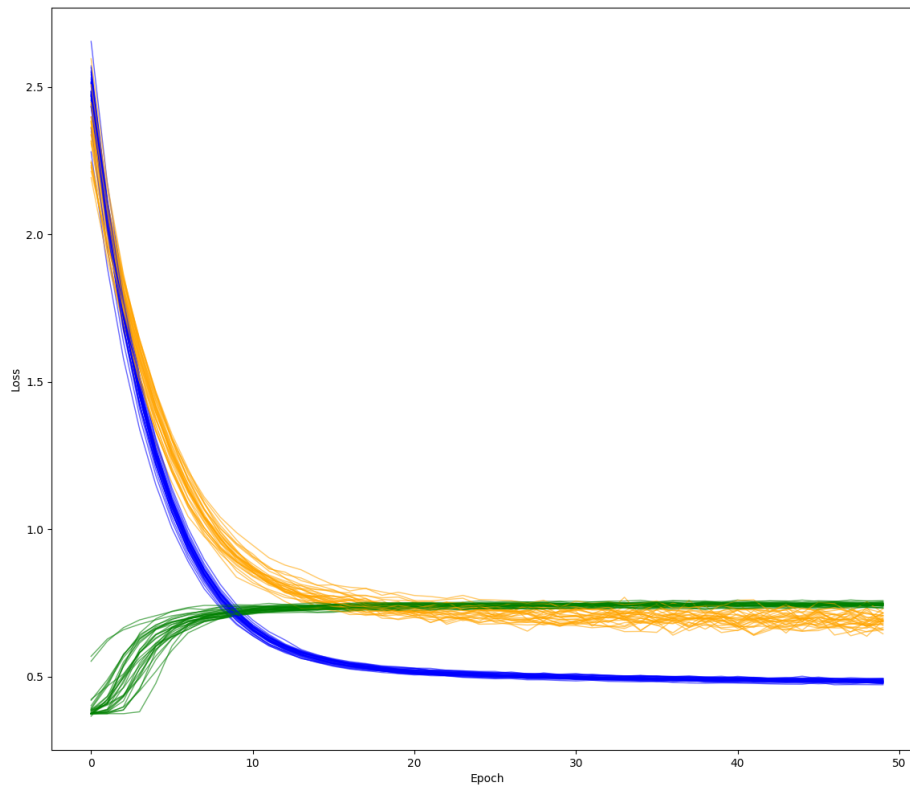
model.evaluate(X_test_norm, Y_test)

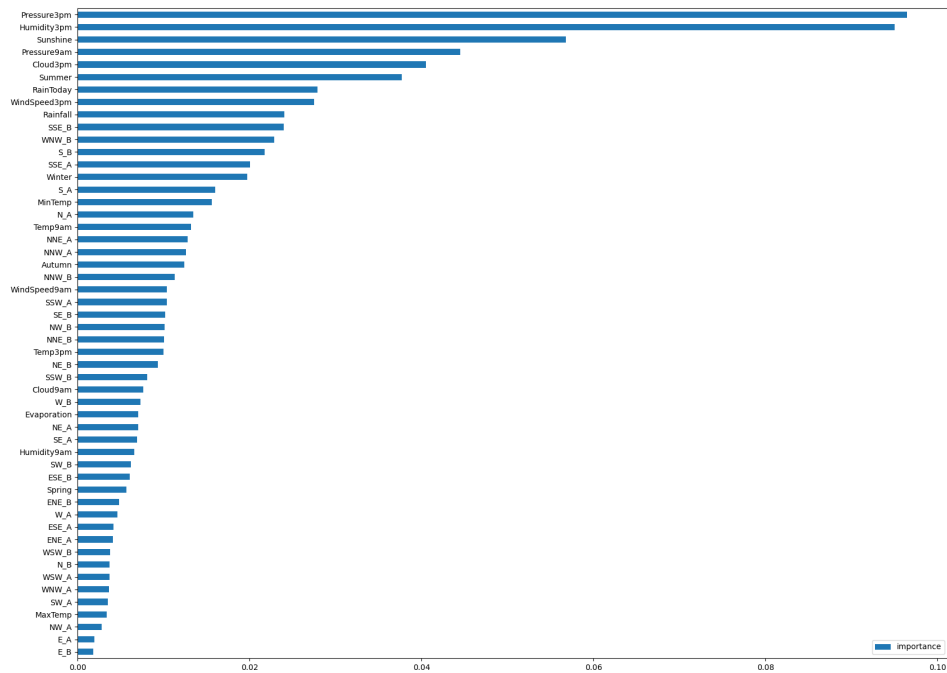
input_tensor = tf.convert_to_tensor(X_train_norm)

with tf.GradientTape() as tape:
    tape.watch(input_tensor)
    output = model(input_tensor)

gradients = tape.gradient(output, input_tensor)
feat_importance = np.mean(np.abs(gradients.numpy()), axis = 0)

```





And the aggregate loss and accuracy data for this new model,

```
[15]: aggregate = pd.read_csv(r"model data\regularised\model_eval.txt")
print(aggregate, "\n")
print("Mean loss:", aggregate["loss"].mean())
print("Mean accuracy:", aggregate["binary_accuracy"].mean())
```

	loss	binary_accuracy
0	0.678191	0.838694
1	0.665628	0.838860
2	0.672140	0.842193
3	0.665285	0.842526
4	0.661624	0.837194
5	0.660761	0.842193
6	0.668548	0.837194
7	0.673919	0.832028
8	0.664702	0.841193
9	0.669644	0.834528
10	0.640113	0.850525
11	0.680263	0.830028
12	0.680238	0.830528
13	0.683306	0.830362
14	0.681786	0.828029
15	0.667525	0.851025

16	0.686789	0.838860
17	0.660936	0.839693
18	0.659312	0.842360
19	0.719382	0.816697
20	0.682386	0.835527
21	0.674532	0.843693
22	0.664147	0.841193
23	0.698843	0.827362
24	0.681566	0.844526
25	0.657982	0.841860
26	0.689237	0.824696
27	0.666993	0.836361
28	0.675462	0.831528
29	0.669244	0.836861

Mean loss: 0.6733495414257049

Mean accuracy: 0.8369438449541727

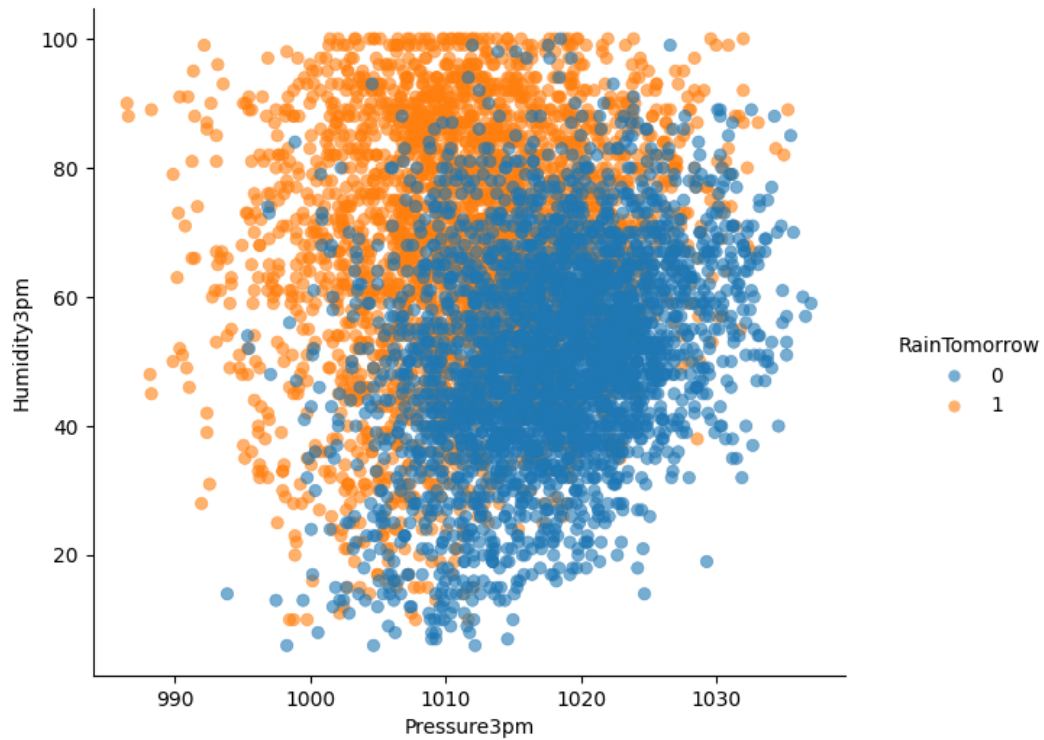
We can see that the loss is reduced significantly, and the mean accuracy increased.

1.0.6 Result Analysis and Further Models

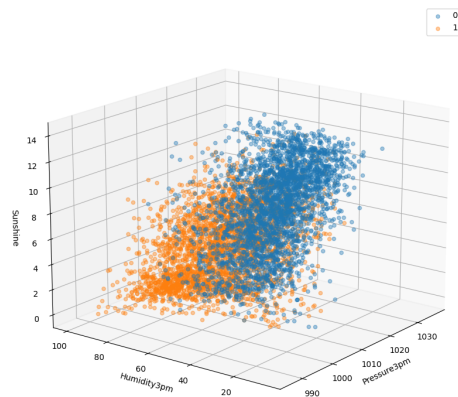
From the plotted graphs, we can see that firstly, reducing the learning rate has caused the fluctuation in training/validation loss to decrease, smoothing out the curves. Similarly, much of the features have a significantly lower contribution/importance to the final prediction. The top features are also quite interesting.

1. Pressure at 3pm
2. Humidity at 3pm
3. Sunshine
4. Pressure at 9am
5. Cloud at 3pm

These features make the most sense in determining the probability of rain the next day. Taking pressure and humidity as an example, low atmospheric pressure can cause air from neighbouring areas to rush in, forcing the air to rise up and condense, forming clouds. High humidity increases the amount of water vapour that condenses, contributing to the likelihood of rain as well. If we take the two most prominent features, Pressure and Humidity at 3pm, and plot a classification scatter plot, we see that there is a significant overlap in the middle as expected, as these two are not the only features at play, so there will not be a clear linear division between the two clusters. However, towards the extremes, such as high humidity and low pressure, versus high pressure and low humidity, we see a clear domination by either classification as expected.



We can attempt to further separate/minimise the overlap by introducing a third dimension to the plot. We choose Sunshine as the third feature as it is the third most “important” feature.



From this, we can see that the “Sunshine” feature skews the datapoints, with low sunshine increasing the likelihood of there being rain the next day, while more sunshine decreases the chance of rain. With the dataset narrowed down to three major features, we attempt to use other classification methods on these three features, namely K-Nearest Neighbours, Decision Tree, Random Forest, and

Support Vector Machines through the scikit-learn library. We use the `GridSearchCV()` method to browse the optimum hyperparameters for each of the model, and pass them through a k=10 fold cross validation.

```
[ ]: import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

data = pd.read_csv("weatherAUSclean.csv")

export_dir = r"model data"

loc = ["MelbourneAirport", "Melbourne", "Watsonia", "Portland", "MountGambier"]

location_data = data[data["Location"].isin(loc)] #["Pressure3pm",
↳ "Humidity3pm", "RainToday", "RainTomorrow"]

train_pool, test_pool = DataSplitter(location_data, n_yes = None, want_test =
↳ True) #train pool now has equal split of yes/no rain

feature_labels = (train_pool.drop(columns = ["Date", "Location",
↳ "RainTomorrow"])).columns

X_train = train_pool.drop(columns = ["Date", "Location", "RainTomorrow"])

Y_train = train_pool["RainTomorrow"]

adapted_normaliser, X_train_norm = DataNormaliser(X_train, normaliser = None)

X_train_norm = X_train_norm[["Pressure3pm", "Humidity3pm", "RainToday"]]

#knn

initK = int(np.sqrt(len(train_pool)))
n = 10
step = 10
K_cycle = np.concatenate((np.arange(initK-n*step, initK, step), np.
↳ arange(initK, initK+n*step, step)))
K_cycle = K_cycle[K_cycle > 0]

parameters = {'n_neighbors': list(K_cycle),
```

```

        'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
        'p': [1,2]}

KNN = KNeighborsClassifier()
knn_cv = GridSearchCV(KNN, parameters, cv=10, verbose = 2)
knn_cv.fit(X_train_norm, Y_train)
with open(export_dir + r"\model_mash.txt", "a+") as file:
    file.write("knn")
    file.write(f"{knn_cv.best_params_},{knn_cv.best_score_}\n")

#dec tree
parameters = {'criterion': ['gini', 'entropy'],
              'splitter': ['best', 'random'],
              'max_depth': [2*n for n in range(1,10)],
              'max_features': ['sqrt', 'log2']}

tree = DecisionTreeClassifier()
tree_cv = GridSearchCV(tree,parameters,cv=10, error_score='raise', verbose = 2)
tree_cv.fit(X_train_norm, Y_train)
with open(export_dir + r"\model_mash.txt", "a+") as file:
    file.write("dectree")
    file.write(f"{tree_cv.best_params_},{tree_cv.best_score_}\n")

#svm
parameters = {
    'C': [0.01,0.1,1,10],
    'kernel' : ["linear","rbf","sigmoid"],
    'degree' : [1,3,5,7],
    'gamma' : [0.01,1,10,500]
}
svm = SVC()
svm_cv = GridSearchCV(svm,parameters,cv=10, verbose = 3)
svm_cv.fit(X_train_norm, Y_train)
with open(export_dir + r"\model_mash.txt", "a+") as file:
    file.write("svm")
    file.write(f"{svm_cv.best_params_},{svm_cv.best_score_}\n")

#random forest
param_grid = {
    'n_estimators': [200, 500],
    'max_features': ['sqrt', 'log2'],
    'max_depth' : [4,5,6,7,8],
    'criterion' :['gini', 'entropy']
}
rfc=RandomForestClassifier()
rfc_cv = GridSearchCV(estimator=rfc, param_grid=param_grid, cv= 10, verbose = 2)

```

```

rfc_cv.fit(X_train_norm, Y_train)
with open(export_dir + r"model_mash.txt", "a+") as file:
    file.write("randfor")
    file.write(f"{rfc_cv.best_params_},{rfc_cv.best_score_}\n")

```

```

[17]: acc_matrix = pd.read_csv("model_data\model_mash.txt", sep = "/")
      print(acc_matrix[["model", "accuracy"]])

```

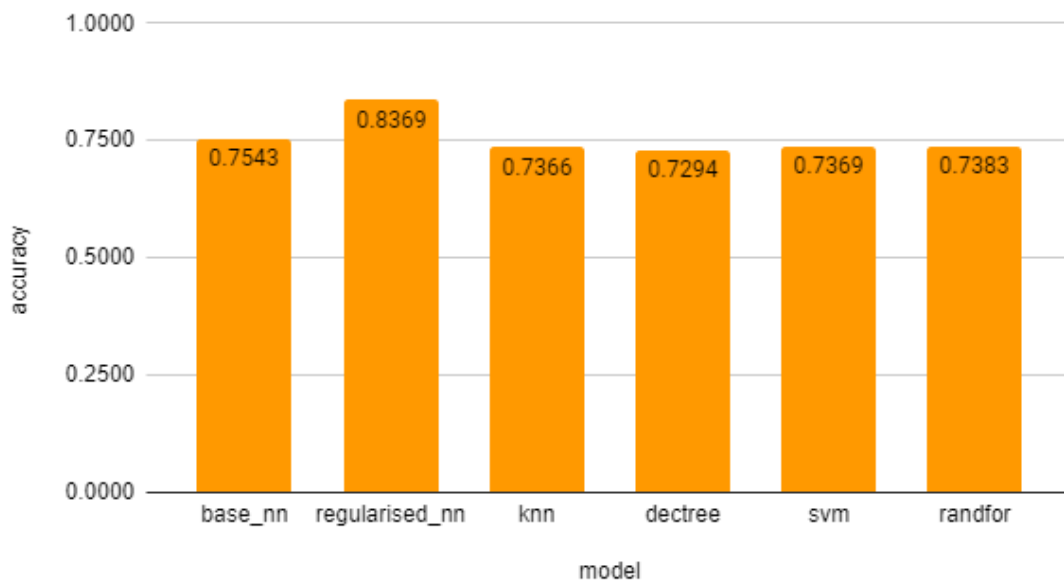
```

      model  accuracy
0      knn   0.736574
1  dectree   0.729438
2      svm   0.736915
3  randfor   0.738274

```

Summarising the accuracies for all the different models, we see that the model with the highest out-of-sample accuracy is the regularised neural network, with a binary accuracy of 83.7%.

Accuracy of models



```

[ ]:

```