Data Analytics and Machine Learning

[**1 - Extrasensory**](#_8whk2bof5vsi)3

[1.1 Improving the Test Set](#_rh3ld1egkf4a) 3

[1.1.1 Analysis](#_k3lglwbl2n93) 3

[1.1.2 Design and Implementation](#_1vr0kk4ccaf2) 3

[1.1.3 Results](#_t7efirufubu5) 3

[1.1.4 Conclusion](#_qoumte84fm3r) 4

[1.2 Validation Data & Increased Training Data](#_vff3elfdfg19) 4

[1.2.1 Analysis](#_6tb9h3b0lpsi) 4

[1.2.2 Design and Implementation](#_svxcr85wt6z0) 4

[1.2.3 Results](#_3io074p6mhm4) 5

[1.2.4 Conclusion](#_x07wswco5k0l) 6

[1.3 Model Selection](#_fvsrvwusvc1g) 6

[1.3.1 Analysis](#_srtz1jj3gxam) 6

[1.3.2 Design and Implementation](#_rqfgnu55r3vn) 6

[1.3.3 Results](#_2a2yxi9hnhw9) 8

[1.3.4 Conclusion](#_4019pl3b15z3) 9

[1.4 Model Testing](#_mn3zqk6t1sqa) 9

[1.4.1 Analysis](#_zazctlr86y2b) 9

[1.4.2 Design and Implementation](#_o99hmeevgrwa) 9

[1.4.3 Results](#_lj90zyakgexi) 10

[1.4.4 Conclusion](#_orefd5yso3qp) 10

[1.5 Predicting Other Actions](#_ule71zj3k4mp) 11

[1.5.1 Analysis](#_lp6k57a3jh6a) 11

[1.5.2 Design and Implementation](#_qjfutq3vb52k) 11

[1.5.3 Conclusion](#_ktbl32liwku5) 11

[**2 - Planet Four**](#_itb7d729slpv)12

[2.1 Train a Model](#_a45vpzi5kggx) 12

[2.1.1 Analysis](#_msa79tew7qsz) 12

[2.1.2 Design and Implementation](#_8b81e436ibq4) 12

[2.1.3 Results](#_b7eh9gc86yhz) 13

[2.1.4 Conclusion](#_mf4j9k8ks2yx) 13

[2.2 Experiments](#_gekpj38ixpgu) 14

[2.2.1 Analysis](#_rcvoc3gxg5ww) 14

[2.2.2 Design and Implementation](#_m65wwaxz9i8u) 14

[2.2.3 Conclusion](#_cik08a6dk0gk) 16

[**Appendix**](#_lpspghef3pgy)16

[Appendix A - sensor\_starter.ipynb](#_87hmbfamsf2l) 16

[Appendix B - planetfour.ipynb](#_jhffu1handov) 24

# 

# 1 - Extrasensory

## 1.1 Improving the Test Set

### 1.1.1 Analysis

The baseline logistic regression model provided in the starter code is initially only tested with a single user. This is not ideal as it would be expected that for a test set with only one sample there will be high variance in test metrics depending on the sample that is present. The goal of this activity is to measure and compare the mean and variance of the evaluation by test sets with a single sample and try to reduce the variance by using multiple sample test sets.

### 1.1.2 Design and Implementation

It was important when conducting this experiment to ensure that samples were exclusive between test sets, i.e. no sample should exist in more than one set. This was done by turning the samples into a list “users”. The model was fitted with the last sample in the list. The test sets with one user used the 1st 5 samples while the multi-sample sets used users 6 to 30, with each set containing 5 users. The metrics being measured were accuracy and balanced accuracy.

### 1.1.3 Results

The mean and variance of the metrics produced by the single sample test sets were the following:

|  |
| --- |
| Evaluation of accuracies of 5x1 test sets  Mean: 0.904229  Variance: 0.004208  Evaluation of balanced accuracies of 5x1 test sets  Mean: 0.696049  Variance: 0.005073 |

Figure 1.1.3a - mean and variance of metrics for single sample test sets

While the test sets with 5 samples each had the following mean and variance:

|  |
| --- |
| Evaluation of accuracies of 5x5 test sets  Mean: 0.927503  Variance: 0.000004  Evaluation of balanced accuracies of 5x5 test sets  Mean: 0.675370  Variance: 0.000155 |

Figure 1.1.3b - mean and variance of metrics for multi-sample test sets

### 1.1.4 Conclusion

As can be seen in figures 1.1.3a and 1.1.3b, the variance of the multi-user test set metrics were orders of magnitude lower than that of the single user evaluations. This was the goal of the activity and the expected outcome.

## 1.2 Validation Data & Increased Training Data

### 1.2.1 Analysis

The next exercise involves two parts; 1) introducing a validation set and 2) combining a number of users to create a larger training set. The purpose of having a validation set is to help evaluate the model during the model selection phase. Hyperparameters are adjusted and the model is evaluated using the validation set to see how the model compares to others. Using a larger training set improves the chances of producing a model with a good fit - one that generalises well and that can adequately solve the problem.

### 1.2.2 Design and Implementation

It made sense to perform the data split that creates the training, validation and test sets in a single function. This is because it produces readable code and helps prevent contamination of any of the sets.

|  |
| --- |
| def split\_data(users, train\_split, val\_split):  n\_users = len(users)  n\_train = round(n\_users \* train\_split)  n\_val = round(n\_users \* val\_split)  n\_test = n\_users - n\_train - n\_val  print(f"Total samples in dataset: {n\_users}")  print(f"Samples in train set: {n\_train}")  print(f"Samples in validation set: {n\_val}")  print(f"Samples in test set: {n\_test}")  i = 0  train\_dfs = []  while i < n\_train:  df = load\_data\_for\_user(users[i].split(".")[0])  train\_dfs.append(df)  i+=1  val\_dfs = []  while i < n\_train + n\_val:  df = load\_data\_for\_user(users[i].split(".")[0])  val\_dfs.append(df)  i+=1  test\_dfs = []  while i < n\_train + n\_val + n\_test:  df = load\_data\_for\_user(users[i].split(".")[0])  test\_dfs.append(df)  i+=1  return pd.concat(train\_dfs), pd.concat(val\_dfs), pd.concat(test\_dfs)  train\_set, val\_set, test\_set = split\_data(users, 0.6, 0.2) |

Figure 1.2.2a - split\_data function

The function “split\_data” was written which takes in a list of users, a float representing the ratio of training data, and a float for the ratio of validation data. It iteratively creates lists of dataframes for the train, validation and test sets and then returns the concatenation of each respective set. A split of 60% training data, 20% validation data and 20% test data was chosen. This should leave an adequate amount of data for training while also ensuring that variance for validation and testing is low.

|  |
| --- |
| Total samples in dataset: 60  Samples in train set: 36  Samples in validation set: 12  Samples in test set: 12 |

Figure 1.2.2b - terminal output from split\_data function

### 1.2.3 Results

A new model was trained using the larger train set and it was subsequently evaluated with the new validation set.

|  |
| --- |
| Fitting model...  194965 examples with 15091.0 positives  Training accuracy: 0.9343  Evaluating model...  3879 examples with 190.0 positives  Test accuracy: 0.9371  Balanced accuracy: 0.6254 |

Figure 1.2.3a - accuracy and balanced accuracy of model with increased training data

A training accuracy of 93.43% was achieved and the validation accuracy was 93.71%. However the balanced accuracy from the validation set was only 62.54%.

### 1.2.4 Conclusion

Compared to the average accuracies and balanced accuracies recorded earlier (figures 1.1.3a and 1.1.3b) for the training set with a single user, the accuracy has slightly increased but the balanced accuracy has gone down. This is likely attributed to an increase in the number of samples in the validation set, as the balanced accuracy appears to trend downwards as the number of test samples increases. The reason normal accuracy isn’t the best metric for model evaluation is because if you have a test set with 90 samples of class A and 10 samples of class B, the accuracy will be 90% by always just giving a prediction of class A. Balanced accuracy is a better indicator of the model performance because it considers unbalanced datasets.

## 1.3 Model Selection

### 1.3.1 Analysis

Now that there is a baseline model and appropriate data split, the next step is to try and produce a better model by modifying the hyperparameters of the logistic regression model, or by trialling alternative models.

### 1.3.2 Design and Implementation

It makes sense to first experiment with changing the hyperparameters of the baseline linear regression such as the C parameter. Then a number of other models will be tested. Increasing the C parameter to 4 from 1 yielded the following accuracies from the logistic regression model:

|  |
| --- |
| Logistic Regression  Fitting model...  194965 examples with 15091.0 positives  Training accuracy: 0.9343  Evaluating model...  3879 examples with 190.0 positives  Test accuracy: 0.9370  Balanced accuracy: 0.6246 |

Figure 1.3.2a - logistic regression model with C=4

This marginally reduced both the accuracy and balanced accuracy in the validation set. Decreasing the C parameter to 0.01 yielded the following accuracies:

|  |
| --- |
| Logistic Regression  Fitting model...  194965 examples with 15091.0 positives  Training accuracy: 0.9342  Evaluating model...  66006 examples with 4259.0 positives  Validation accuracy: 0.9393  Balanced accuracy: 0.6559 |

Figure 1.3.2b - logistic regression classifier with C=0.01

Training accuracy decreased marginally but validation accuracy improved and the validation balanced accuracy increased by around 3%.

Next a decision tree model was fitted.

|  |
| --- |
| Decision Tree  Fitting model...  194965 examples with 15091.0 positives  Training accuracy: 1.0000  Evaluating model...  66006 examples with 4259.0 positives  Validation accuracy: 0.7459  Balanced accuracy: 0.5811 |

Figure 1.3.2c - decision tree classifier

The training accuracy was at 100% while the validation accuracy and balanced accuracy were 74% and 58%. This indicates that the decision tree was overfitting. To counteract this the maximum depth of the tree was set to 10 and this led to improved accuracies:

|  |
| --- |
| Decision Tree  Fitting model...  194965 examples with 15091.0 positives  Training accuracy: 0.9414  Evaluating model...  66006 examples with 4259.0 positives  Validation accuracy: 0.9294  Balanced accuracy: 0.6643 |

Figure 1.3.2d - decision tree classifier with max\_depth=10

Now the performance of the decision tree was roughly on par with the logistic regression classifier. The result could be further improved by using the random forest ensemble method, which uses a collection of randomised decision trees and makes predictions from the aggregate. The max depth of the random forest was set to 10 based on the results of the previous experiment. The number of trees was set to 50.

|  |
| --- |
| Random Forest  Fitting model...  194965 examples with 15091.0 positives  Training accuracy: 0.9436  Evaluating model...  66006 examples with 4259.0 positives  Validation accuracy: 0.9468  Balanced accuracy: 0.6824 |

Figure 1.3.2e - random forest classifier

Now the validation accuracy was slightly better than the improved logistic regression classifier, and the balanced accuracy was around 3% better.

One last model would be trialled, a gaussian naive bayes classifier. This produced good validation results:

|  |
| --- |
| Gaussian Naive Bayes  Fitting model...  194965 examples with 15091.0 positives  Training accuracy: 0.8819  Evaluating model...  66006 examples with 4259.0 positives  Validation accuracy: 0.8327  Balanced accuracy: 0.7443 |

Figure 1.3.2f - gaussian naive bayes classifier

Both the training and validation accuracy were lower than the other models, but the validation balanced accuracy was considerably higher making this model a good candidate.

### 1.3.3 Results

|  |  |  |  |
| --- | --- | --- | --- |
| Classifier | Training Accuracy | Validation Accuracy | Validation Balanced Accuracy |
| Logistic Regression C=1 | 0.9343 | 0.9371 | 0.6254 |
| Logistic Regression C=4 | 0.9343 | 0.9370 | 0.6246 |
| Logistic Regression C=0.01 | 0.9342 | 0.9393 | 0.6559 |
| Decision Tree max\_depth=None | 1.0000 | 0.7459 | 0.5811 |
| Decision Tree max\_depth=10 | 0.9414 | 0.9294 | 0.6643 |
| Random Forest n\_trees=50, max\_depth=10 | 0.9436 | 0.9468 | 0.6824 |
| Gaussian Naive Bayes | 0.8819 | 0.8327 | 0.7443 |

The above table summarises the results of the model selection experiments.

### 1.3.4 Conclusion

The logistic regression models were only able to achieve a balanced accuracy of 65.5% and this was achieved by lowering the C value which increases the amount of regularisation. The decision tree classifiers are prone to overfitting, especially in unbalanced datasets and this was rectified by limiting the maximum depth of the tree. Using a random forest rather than a single decision tree got the balanced accuracy up to 68.2%. The highest balanced accuracy of 74.4% was achieved by using a gaussian naive bayes classifier. In the next section, the two best candidates; the naive bayes and random forest will be further evaluated with the holdout test set.

## 1.4 Model Testing

### 1.4.1 Analysis

Previously when evaluating the classifiers, a validation set was being used. However by using the validation score to tune the hyperparameters, the model is implicitly being fit to the validation set. This is why having a separate holdout test set is important for getting the true final performance evaluation of the model.

### 1.4.2 Design and Implementation

The metrics included in the final performance evaluation will be:

* Accuracy - The ratio of correct predictions.
* Recall - The ratio of actual positive samples found.
* Precision - The ratio of negative samples correctly predicted as negative.
* F1 score - Weighted combination of recall and precision, needs both values to be high to achieve a good score.
* Balanced accuracy - Accuracy that takes into account unbalanced dataset; the average of recall for each class.

Even though the naive bayes appeared to be the best classifier going by balanced accuracy, the random forest will also be evaluated in case more information is revealed by the additional metrics. The evaluation function use the sklearn built-in metrics.

def get\_model\_perf(model, test\_set, acc\_sensors, target\_column):

X\_test, y\_test = get\_features\_and\_target(test\_set, acc\_sensors, target\_column)

print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')

X\_test = imputer.transform(scaler.transform(X\_test))

print(f'Test accuracy: {model.score(X\_test, y\_test):0.4f}')

y\_pred = model.predict(X\_test)

print(f'Recall: {metrics.recall\_score(y\_test, y\_pred):0.4f}')

print(f'Precision: {metrics.precision\_score(y\_test, y\_pred):0.4f}')

print(f'Balanced accuracy: {metrics.balanced\_accuracy\_score(y\_test, y\_pred):0.4f}')

print(f'F1 score: {metrics.f1\_score(y\_test, y\_pred):0.4f}')

Figure 1.4.2a - evaluation function

### 1.4.3 Results

The scores for the naive bayes model were:

|  |
| --- |
| Testing Naive Bayes Classifier...  3879 examples with 190.0 positives  Test accuracy: 0.8412  Recall: 0.7251  Precision: 0.2377  Balanced accuracy: 0.7869  F1 score: 0.3581 |

Figure 1.4.3a - naive bayes final performance

While the random forest yielded:

|  |
| --- |
| Testing Random Forest Classifier...  3879 examples with 190.0 positives  Test accuracy: 0.9573  Recall: 0.4788  Precision: 0.7298  Balanced accuracy: 0.7336  F1 score: 0.5782 |

Figure 1.4.3b - random forest final performance

### 1.4.4 Conclusion

During the model selection phase it seemed like the naive bayes was the better model due to its higher balanced accuracy. However with the additional metrics, it was revealed that while the recall is decent, precision is very low meaning that there are a lot of false positive predictions. The low precision means that the F1 score is low.

The random forest classifier on the other hand is a more well rounded model with mediocre recall and decent precision and balanced accuracy. The model that should ultimately be chosen would depend on the application. If having fewer false negative predictions is more important (e.g. medical testing) then the naive bayes model is more suitable. If having fewer false positive predictions is desirable then the random forest classifier should be selected.

## 1.5 Predicting Other Actions

### 1.5.1 Analysis

The goal of this activity is to work with the optimal model (subjectively the random forest), and try to predict some actions other than walking. This will help demonstrate the robustness of the model.

### 1.5.2 Design and Implementation

The implementation was relatively trivial, it required changing the value of ‘target\_column’ (in this case to 'label:BICYCLING') and potentially adding or removing features that are more relevant to the new activity. The first new activity to be predicted was bicycling. The features were unchanged as acceleration values should be highly correlated with cycling activity. The model was retrained and tested:

|  |
| --- |
| Testing Random Forest Classifier...  3879 examples with 1145.0 positives  Test accuracy: 0.9905  Recall: 0.6174  Precision: 0.9561  Balanced accuracy: 0.8084  F1 score: 0.7503 |

Figure 1.5.2a - results for cycling activity classification

### 1.5.3 Conclusion

The random forest classifier performed even better for bicycling activity than walking. It is assumed that this is because the acceleration features are more pronounced when the user is cycling. It demonstrates that the model is suitable for other activities.

# 2 - Planet Four

## 2.1 Train a Model

### 2.1.1 Analysis

The goal of this exercise is to train a baseline model for the classification of Martian surface features. The baseline will use the ResNet-50 convolutional neural network (CNN) architecture. The data is already split into train, validation and test sets.

### 2.1.2 Design and Implementation

First of all, the CNN was trained for 5 epochs.

|  |
| --- |
| 100%|██████████| 376/376 [01:22<00:00, 4.58it/s]  [01] train loss: 0.3947 valid loss: 0.4046 fan acc: 0.7877 blotch acc: 0.8356 both acc: 0.6529  100%|██████████| 376/376 [01:22<00:00, 4.55it/s]  [02] train loss: 0.3525 valid loss: 0.3947 fan acc: 0.8019 blotch acc: 0.8416 both acc: 0.6762  100%|██████████| 376/376 [01:22<00:00, 4.55it/s]  [03] train loss: 0.3194 valid loss: 0.4010 fan acc: 0.8042 blotch acc: 0.8353 both acc: 0.6709  100%|██████████| 376/376 [01:22<00:00, 4.53it/s]  [04] train loss: 0.2863 valid loss: 0.4094 fan acc: 0.7986 blotch acc: 0.8390 both acc: 0.6679  100%|██████████| 376/376 [01:23<00:00, 4.49it/s]  [05] train loss: 0.2520 valid loss: 0.4316 fan acc: 0.7990 blotch acc: 0.8338 both acc: 0.6623 |

Figure 2.1.2a - baseline fit for 5 epochs

Two functions were written to plot the loss and accuracy over epochs. Being able to visualise loss over time makes it easier to identify overfitting in the model.

|  |
| --- |
| def plot\_loss(avg\_train\_losses, avg\_valid\_losses):  validation\_losses = pd.DataFrame(avg\_valid\_losses)  training\_losses = pd.DataFrame(avg\_train\_losses)  plt.plot(validation\_losses)  plt.plot(training\_losses)  plt.ylabel("loss")  plt.xlabel("epoch")  plt.legend(["validation", "training"])  plt.show()  def plot\_accuracy(valid\_accuracies):  accuracies = pd.DataFrame(valid\_accuracies)  plt.plot(accuracies)  plt.ylabel("accuracy")  plt.xlabel("epoch")  plt.legend(["fan", "blotch", "both"])  plt.show() |

Figure 2.1.2a - plot functions

The option to save checkpoints of the model after each epoch was enabled. This means that it is possible to go back and select the most optimal version of the model. Finally the model was trained for another 5 epochs and the loss and accuracy were plotted.

### 2.1.3 Results

The baseline model that was trained for 10 epochs had the following loss and accuracy curves:

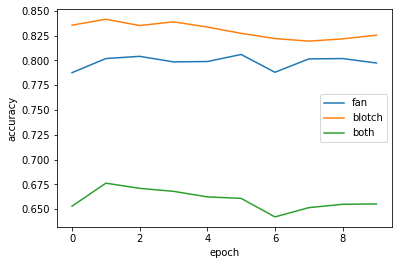
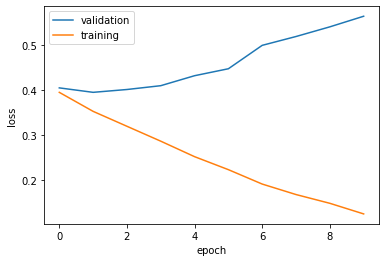


Figure 2.1.3a - loss and accuracy plots of baseline CNN

As can be seen in the loss graph in figure 2.1.3a, the training loss converges towards zero steadily while the validation loss almost immediately increases. This is indicative of a strongly overfitting model. The accuracy fluctuates a bit and overall does not increase with more training. The final accuracies were fan=79.9%, blotch=83.38%, both=66.23%.

### 2.1.4 Conclusion

The baseline model is overfitting and will need to be improved. In the next section improvements will be investigated such as adding regularisation and experimenting with various hyperparameters.

## 2.2 Experiments

### 2.2.1 Analysis

There are a number of improvements that can be made to the baseline CNN such as trying different optimisers, regularisation, data augmentation, and tuning the other hyperparameters. The biggest problem with the baseline model is the strong overfitting.

### 2.2.2 Design and Implementation

The approach taken was to start with the most easily implementable enhancements. The optimiser was changed from SGD to Adam. Adam is an optimiser that takes into account both first and second order moments. A new model was trained for 10 epochs using the Adam optimiser:

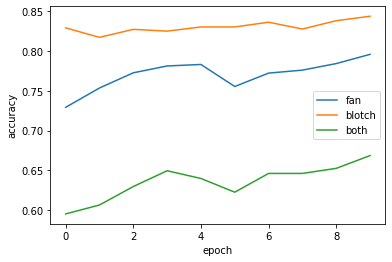
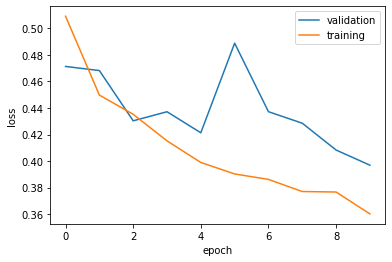


Figure 2.2.2a - loss and accuracy plots of CNN with Adam optimiser

It appears that the model is not overfitting as strongly as before as the validation loss is converging. The accuracies of this model at epoch 10 were fan=79.6%, blotch=84.38%, both=66.87%. There was not much of an improvement in accuracies but at least overfitting is reduced. This means that the model could be trained for more epochs. Training for an additional 5 epochs produced the following loss and accuracy curves:

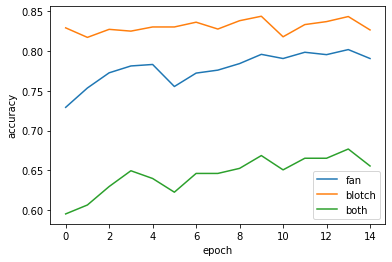
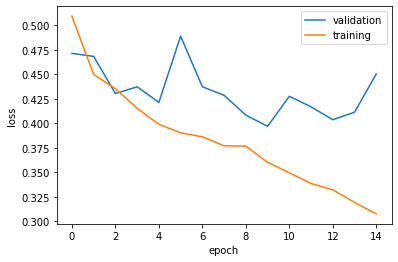


Figure 2.2.2b - loss and accuracy plots of CNN with Adam optimiser, extra 5 epochs

As can be seen in figure 2.2.2b, the model starts to overfit after the 10th epoch and the accuracy does not significantly improve. Another way to reduce overfitting is data augmentation. This is the practice of artificially increasing the number os samples in the training set by applying random transformations such as horizontal and vertical flips, rotations, shears and crops. Some data augmentation is already present in the baseline in the form of horizontal flips. When using data augmentation it is important to consider the type of data present. For example if trying to classify images of people standing up, it probably wouldn’t make sense to rotate the images by a large extent. But since the PlanetFour dataset is made up of aerial images, any range of rotations, horizontal and vertical flips all make sense. The data augmentation was updated.

train\_transform = transforms.Compose([

transforms.RandomHorizontalFlip(),

transforms.RandomVerticalFlip(),

transforms.RandomRotation(180),

transforms.ToTensor(),

transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))

])

valid\_transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))

])

And a new model was trained for 35 epochs as it was expected that convergence would be slower.

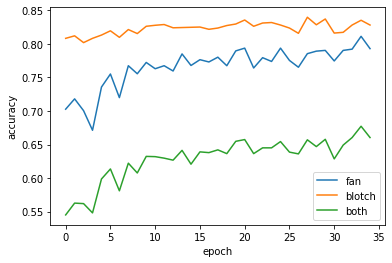
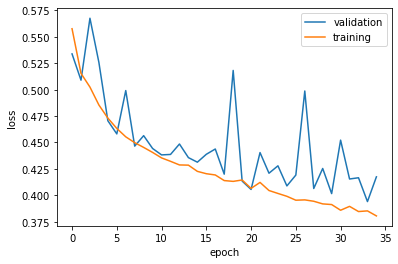


Figure 2.2.2c - loss and accuracy plots of CNN with additional data augmentation

Figure 2.2.2c shows the loss and accuracy curves for this experiment. Adding additional data augmentation made training the model take more epochs. Overfitting was further reduced and the best model so far was produced at epoch 34 with fan=81.1%, blotch=83.5%, both=67.7%.

### 2.2.3 Conclusion

It proved difficult to increase the accuracy over the baseline model and only marginal gains were made here. However it was possible to reduce overfitting by changing the optimiser to Adam and by increasing the number of training samples with data augmentation. If there were more time and resources to work on this exercise I would add even more data augmentation (e.g. contrast, zoom) and train for a large number of epochs. Additionally I would have added a dropout layer between the last layer of ResNet 50 and the new dense layer. This would provide some regularisation but it would be expected that even more epochs would be needed to converge.

# Appendix

## Appendix A - sensor\_starter.ipynb

|  |
| --- |
| import os  import pandas as pd  import numpy as np  from pathlib import Path  import sklearn.metrics as metrics  import matplotlib.pyplot as plt  from sklearn.linear\_model import LogisticRegression  from sklearn import svm  from sklearn import tree  from sklearn.ensemble import RandomForestClassifier  from sklearn.preprocessing import StandardScaler  from sklearn.impute import SimpleImputer  from sklearn.naive\_bayes import GaussianNB  from sklearn.ensemble import VotingClassifier  from google.colab import drive, files  ## Location of the .csv.gz files  drive.mount("/content/gdrive", force\_remount=True)  data\_dir = '/content/gdrive/MyDrive/Data-Analysis-&-Machine-Learning/data/ExtraSensory/ExtraSensory.per\_uuid\_features\_labels'  users = os.listdir(data\_dir)  print(users)  ## Some utility functions  The first one loads a pandas dataframe given a user UUID. The second one extracts specified feature columns $X$ and target column $y$ from a dataframe and converts these to numpy.  def get\_user\_id(user):  return user.split(".")[0]  def load\_data\_for\_user(uuid):  return pd.read\_csv(data\_dir + "/" + (uuid + '.features\_labels.csv.gz'))  def get\_features\_and\_target(df, feature\_names, target\_name):    # select out features and target columns and convert to numpy  X = df[feature\_names].to\_numpy()  y = df[target\_name].to\_numpy()    # remove examples with no label  has\_label = ~np.isnan(y)  X = X[has\_label,:]  y = y[has\_label]  return X, y  ## Load in some data  Load in the data for a user and display the first few rows of the dataframe  df = load\_data\_for\_user('0A986513-7828-4D53-AA1F-E02D6DF9561B')  # df = load\_data\_for\_user(get\_user\_id(users[-1]))  df.head()  ## What columns are available?  print(df.columns.to\_list())  print(len(df.columns.to\_list()))  ## Feature selection  The columns that start with `label:` correspond to potential y values. Let's look at using the accelerometer features. These start with `raw\_acc:` and `watch\_acceleration:`  # acc\_sensors = [s for s in df.columns if  # s.startswith('raw\_acc:') or  # s.startswith('watch\_acceleration:')]  # # target\_column = 'label:FIX\_walking'  # target\_column = 'label:FIX\_walking'  ## Extract our training data  X\_train, y\_train = get\_features\_and\_target(df, acc\_sensors, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  ## Preprocessing  We want to make the learning problem easier by making all columns have a mean of zero and a standard deviation of one. There are also lots of missing values in this dataset. We'll use mean imputation here to get rid of them. Since our data is scaled to have zero mean, this will just zero out missing values.  scaler = StandardScaler()  imputer = SimpleImputer(strategy='mean')  X\_train = scaler.fit\_transform(X\_train)  X\_train = imputer.fit\_transform(X\_train)  ## Fitting a model  Let's fit a logistic regression model to this user. We can then test it's predictive power on a different user  clf = LogisticRegression(solver='liblinear', max\_iter=1000, C=1.0)  clf.fit(X\_train, y\_train)  ## Training accuracy  Let's see the accuracy on the training set. The score function can be used to do this:  print(f'Training accuracy: {clf.score(X\_train, y\_train):0.4f}')  Looks like the model can fit the training data reasonably well anyway. But this says nothing about how well it will generalize to new data. The dataset is also unbalanced, so this figure may be misleading. How accurate would we be if we just predicted zero each time?  1 - y\_train.sum() / y\_train.shape[0]  Oh wow. Our model may not be that great after all. Let's try to calculate balanced accuracy, which should better reflect how well the model does on the training data  y\_pred = clf.predict(X\_train)  print(f'Balanced accuracy (train): {metrics.balanced\_accuracy\_score(y\_train, y\_pred):0.4f}')  ## Testing the model  Ok, it seems our model has fit the training data well. How well does it perform on unseen test data? Let's load the data in for a different user.  df\_test = load\_data\_for\_user('11B5EC4D-4133-4289-B475-4E737182A406')  X\_test, y\_test = get\_features\_and\_target(df\_test, acc\_sensors, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  We also need to preprocess as before. \*\*Note\*\*: we are using the scaler and imputer fit to the training data here. It's very important that you do not call `fit` or `fit\_transform` here! Think about why.  X\_test = imputer.transform(scaler.transform(X\_test))  ## Test accuracy  print(f'Test accuracy: {clf.score(X\_test, y\_test):0.4f}')  y\_pred = clf.predict(X\_test)  print(f'Balanced accuracy (train): {metrics.balanced\_accuracy\_score(y\_test, y\_pred):0.4f}')  # Improving the test set  # Train models and return list of accuracies for n users  def get\_accuracies(model, test\_dfs):  accuracies = []  bal\_accuracies = []  for df in test\_dfs:  X\_test, y\_test = get\_features\_and\_target(df, acc\_sensors, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  X\_test = imputer.transform(scaler.transform(X\_test))  print(f'Test accuracy: {clf.score(X\_test, y\_test):0.4f}')  y\_pred = clf.predict(X\_test)  print(f'Balanced accuracy (train): {metrics.balanced\_accuracy\_score(y\_test, y\_pred):0.4f}')  accuracies.append(clf.score(X\_test, y\_test))  bal\_accuracies.append(metrics.balanced\_accuracy\_score(y\_test, y\_pred))  return accuracies, bal\_accuracies  test\_dfs = []  for user in users[:5]:  test\_dfs.append(load\_data\_for\_user(get\_user\_id(user)))  accuracies, bal\_accuracies = get\_accuracies(clf, test\_dfs)  # Show mean and variance of accuracies for 5 test sets with 1 user  print()  print("Evaluation of accuracies of 5x1 test sets")  print(f"Mean: {np.mean(accuracies):0.6f}")  print(f"Variance: {np.var(accuracies):0.6f}")  print("\nEvaluation of balanced accuracies of 5x1 test sets")  print(f"Mean: {np.mean(bal\_accuracies):0.6f}")  print(f"Variance: {np.var(bal\_accuracies):0.6f}")  def build\_test\_set(users, slice\_start, slice\_end):  dfs = []  for user in users[slice\_start:slice\_end]:  df = load\_data\_for\_user(user.split(".")[0])  dfs.append(df)  return pd.concat(dfs)  def test\_model(model, test\_set):  X\_test, y\_test = get\_features\_and\_target(test\_set, acc\_sensors, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  X\_test = imputer.transform(scaler.transform(X\_test))  print(f'Test accuracy: {model.score(X\_test, y\_test):0.4f}')  y\_pred = model.predict(X\_test)  print(f'Balanced accuracy: {metrics.balanced\_accuracy\_score(y\_test, y\_pred):0.4f}')  # print(f'F1 score: {metrics.f1\_score(y\_test, y\_pred):0.4f}')  # Build 5 different test sets to demonstrate mean and variance of accuracy on larger sets  test\_set1 = build\_test\_set(users, 5, 10)  test\_set2 = build\_test\_set(users, 10, 15)  test\_set3 = build\_test\_set(users, 15, 20)  test\_set4 = build\_test\_set(users, 20, 25)  test\_set5 = build\_test\_set(users, 25, 30)  dfs = [test\_set1, test\_set2, test\_set3, test\_set4, test\_set5]  accuracies, bal\_accuracies = get\_accuracies(clf, dfs)  # Show mean and variance of accuracies for 5 test sets with 5 users  print()  print("Evaluation of accuracies of 5x5 test sets")  print(f"Mean: {np.mean(accuracies):0.6f}")  print(f"Variance: {np.var(accuracies):0.6f}")  print("\nEvaluation of balanced accuracies of 5x5 test sets")  print(f"Mean: {np.mean(bal\_accuracies):0.6f}")  print(f"Variance: {np.var(bal\_accuracies):0.6f}")  # test\_model(clf, test\_set1)  ### Data Splitting  def split\_data(users, train\_split, val\_split):  n\_users = len(users)  n\_train = round(n\_users \* train\_split)  n\_val = round(n\_users \* val\_split)  n\_test = n\_users - n\_train - n\_val  print(f"Total samples in dataset: {n\_users}")  print(f"Samples in train set: {n\_train}")  print(f"Samples in validation set: {n\_val}")  print(f"Samples in test set: {n\_test}")  i = 0  train\_dfs = []  while i < n\_train:  df = load\_data\_for\_user(users[i].split(".")[0])  train\_dfs.append(df)  i+=1  val\_dfs = []  while i < n\_train + n\_val:  df = load\_data\_for\_user(users[i].split(".")[0])  val\_dfs.append(df)  i+=1  test\_dfs = []  while i < n\_train + n\_val + n\_test:  df = load\_data\_for\_user(users[i].split(".")[0])  test\_dfs.append(df)  i+=1  return pd.concat(train\_dfs), pd.concat(val\_dfs), pd.concat(test\_dfs)  train\_set, val\_set, test\_set = split\_data(users, 0.6, 0.2)  ### Model Selection  # Evaluate model with validation set  def evaluate\_model(model, val\_set, features, target\_column):  X\_test, y\_test = get\_features\_and\_target(val\_set, features, target\_column)  print(f'{y\_test.shape[0]} examples with {y\_test.sum()} positives')  X\_test = imputer.transform(scaler.transform(X\_test))  print(f'Validation accuracy: {model.score(X\_test, y\_test):0.4f}')  y\_pred = model.predict(X\_test)  print(f'Balanced accuracy: {metrics.balanced\_accuracy\_score(y\_test, y\_pred):0.4f}')  # Train logistic regression model  # It was found that reducing the C parameter increased the balanced accuracy and F1 score  def train\_lr\_model(train\_set, features, target\_column, C\_param):  X\_train, y\_train = get\_features\_and\_target(train\_set, features, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  scaler = StandardScaler()  imputer = SimpleImputer(strategy='mean')  X\_train = scaler.fit\_transform(X\_train)  X\_train = imputer.fit\_transform(X\_train)  model = LogisticRegression(solver='liblinear', max\_iter=1000, C=C\_param)  model.fit(X\_train, y\_train)  print(f'Training accuracy: {model.score(X\_train, y\_train):0.4f}')  return model  # Train decision tree model  def train\_dt\_model(train\_set, features, target\_column, max\_depth):  X\_train, y\_train = get\_features\_and\_target(train\_set, features, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  scaler = StandardScaler()  imputer = SimpleImputer(strategy='mean')  X\_train = scaler.fit\_transform(X\_train)  X\_train = imputer.fit\_transform(X\_train)  model = tree.DecisionTreeClassifier(max\_features="sqrt", max\_depth=max\_depth, min\_samples\_split=2)  model.fit(X\_train, y\_train)  print(f'Training accuracy: {model.score(X\_train, y\_train):0.4f}')  return model  # Train random forest model  def train\_rf\_model(train\_set, features, target\_column, n\_trees, max\_depth):  X\_train, y\_train = get\_features\_and\_target(train\_set, features, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  scaler = StandardScaler()  imputer = SimpleImputer(strategy='mean')  X\_train = scaler.fit\_transform(X\_train)  X\_train = imputer.fit\_transform(X\_train)  model = RandomForestClassifier(n\_estimators=n\_trees, max\_features="sqrt", max\_depth=max\_depth, min\_samples\_split=2, min\_samples\_leaf=2)  model.fit(X\_train, y\_train)  print(f'Training accuracy: {model.score(X\_train, y\_train):0.4f}')  return model  # Train naive bayes model  def train\_nb\_model(train\_set, features, target\_column):  X\_train, y\_train = get\_features\_and\_target(train\_set, features, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  scaler = StandardScaler()  imputer = SimpleImputer(strategy='mean')  X\_train = scaler.fit\_transform(X\_train)  X\_train = imputer.fit\_transform(X\_train)  model = GaussianNB()  model.fit(X\_train, y\_train)  print(f'Training accuracy: {model.score(X\_train, y\_train):0.4f}')  return model  # Train and evaluate logistic regression model  features = [s for s in train\_set.columns if  s.startswith('raw\_acc:') or  s.startswith('watch\_acceleration:')]  target\_column = 'label:BICYCLING'  print("Logistic Regression")  print("Fitting model...")  model\_lr = train\_lr\_model(train\_set, features, target\_column, 0.01)  print("\nEvaluating model...")  evaluate\_model(model\_lr, val\_set, features, target\_column)  print("Decision Tree")  print("Fitting model...")  model\_dt = train\_dt\_model(train\_set, features, target\_column, 10)  print("\nEvaluating model...")  evaluate\_model(model\_dt, val\_set, features, target\_column)  print("Random Forest")  print("Fitting model...")  model\_rf = train\_rf\_model(train\_set, features, target\_column, 50, 10)  print("\nEvaluating model...")  evaluate\_model(model\_rf, val\_set, features, target\_column)  print("Gaussian Naive Bayes")  print("Fitting model...")  model\_nb = train\_nb\_model(train\_set, features, target\_column)  print("\nEvaluating model...")  evaluate\_model(model\_nb, val\_set, features, target\_column)  def get\_model\_perf(model, test\_set, features, target\_column):  X\_test, y\_test = get\_features\_and\_target(test\_set, features, target\_column)  print(f'{y\_train.shape[0]} examples with {y\_train.sum()} positives')  X\_test = imputer.transform(scaler.transform(X\_test))  print(f'Test accuracy: {model.score(X\_test, y\_test):0.4f}')  y\_pred = model.predict(X\_test)  print(f'Recall: {metrics.recall\_score(y\_test, y\_pred):0.4f}')  print(f'Precision: {metrics.precision\_score(y\_test, y\_pred):0.4f}')  print(f'Balanced accuracy: {metrics.balanced\_accuracy\_score(y\_test, y\_pred):0.4f}')  print(f'F1 score: {metrics.f1\_score(y\_test, y\_pred):0.4f}')  # print(metrics.confusion\_matrix(y\_test, y\_pred))  # print(metrics.classification\_report(y\_test, y\_pred, digits=3))  print("Testing Random Forest Classifier...")  get\_model\_perf(model\_rf, test\_set, features, target\_column)  print("Testing Naive Bayes Classifier...")  get\_model\_perf(model\_nb, test\_set, features, target\_column) |

## Appendix B - planetfour.ipynb

|  |
| --- |
| ## Planet four image classification  # !tar -xvf '/content/gdrive/MyDrive/ML/data/PlanetFour/planetfour.tar' -C '/content/gdrive/MyDrive/ML/data/PlanetFour/split/'  # !ls '/content/gdrive/MyDrive/ML/data/PlanetFour/split/planetfour'  #!ls '/content/gdrive/MyDrive/ML/data/PlanetFour/'  import numpy as np  import matplotlib.pyplot as plt  import pandas as pd  import os  import torch  import torch.nn.functional as F  import torch.nn as nn  import torch.optim as optim  import torchvision.transforms as transforms  import torchvision.models as models  import sklearn.metrics as metrics  import tqdm  from torch.utils.data import DataLoader  from torchvision.datasets.folder import pil\_loader  from pathlib import Path  from PIL import Image  from google.colab import drive, files  Change the device to "cpu" if you want to train on a CPU instead of a GPU.  device = 'cuda'  print(torch.cuda.device\_count())  print(torch.cuda.get\_device\_name())  drive.mount("/content/gdrive", force\_remount=True)  data\_dir = '/content/gdrive/MyDrive/ML/data/PlanetFour/split/planetfour'  # # Print num samples in each set  # train\_dir = f"{data\_dir}/train"  # valid\_dir = f"{data\_dir}/valid"  # test\_dir = f"{data\_dir}/test"  # print(f"Training samples: {len(os.listdir(train\_dir))}")  # print(f"Validation samples: {len(os.listdir(valid\_dir))}")  # print(f"Test samples: {len(os.listdir(test\_dir))}")  ## Dataset  Here we define a custom Dataset object for the Planet Four data. You can read more about this in the PyTorch documentation: https://pytorch.org/tutorials/beginner/basics/data\_tutorial.html  class PlanetFourDataset(object):  def \_\_init\_\_(self, split='train', transform=None, loader=pil\_loader):  super().\_\_init\_\_()  self.split = split  self.base\_dir = Path(data\_dir)  self.image\_dir = f"{self.base\_dir}/{self.split}"  self.labels\_file = f"{self.base\_dir}/{(split + '.csv')}"  self.labels\_df = pd.read\_csv(self.labels\_file)  self.transform = transform  self.loader = loader    def \_\_getitem\_\_(self, index):  row = self.labels\_df.iloc[index]  filename = f"{self.image\_dir}/{(row.tile\_id + '.jpg')}"  fans = int(row.fans)  blotches = int(row.blotches)  image = self.loader(str(filename))  if self.transform is not None:  image = self.transform(image)  return image, torch.tensor([fans, blotches], dtype=torch.float32)    def \_\_len\_\_(self):  return len(self.labels\_df)  ## Data augmentation  It is standard practice in deep learning to augment the training examples to prevent the network from overfitting. Here I use some standard augmentations such as randomly mirroring the images.  train\_transform = transforms.Compose([  transforms.RandomHorizontalFlip(),  transforms.RandomVerticalFlip(),  transforms.RandomRotation(180),  transforms.ToTensor(),  transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))  ])  valid\_transform = transforms.Compose([  transforms.ToTensor(),  transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))  ])  ## Data loaders  In PyTorch, the data loaders take care of spinning up threads to load batches of data into memory from the dataset object.  train\_set = PlanetFourDataset('train', transform=train\_transform)  valid\_set = PlanetFourDataset('valid', transform=train\_transform)  train\_loader = DataLoader(train\_set, batch\_size=64, shuffle=True)  valid\_loader = DataLoader(valid\_set, batch\_size=64, shuffle=False)  ## Load a pretrained model  Here we'll use ResNet50 model that has been pretrained on ImageNet and replace the final layer with a new one suited to our problem.  model = models.resnet50(pretrained=True)  model.fc = nn.Linear(2048, 2)  model.to(device);  ## Loss  Images can contain fans, blotches, both, or neither. You could treat this as a four class softmax problem, or two binary classification problems. Here I take the latter approach and use a binary cross entropy loss.  criterion = nn.BCEWithLogitsLoss()  ## Optimizer  Stochastic gradient descent with momentum  # optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9, weight\_decay=1e-4)  optimizer = optim.Adam(model.parameters())  ## Training and validation functions  avg\_train\_losses = []  avg\_valid\_losses = []  valid\_accuracies = []  def train\_for\_epoch(optimizer):  model.train()  train\_losses = []  for batch, target in tqdm.tqdm(train\_loader):  # data to GPU  batch = batch.to(device)  target = target.to(device)  # reset optimizer  optimizer.zero\_grad()  # forward pass  predictions = model(batch)  #breakpoint()  # calculate loss  loss = criterion(predictions, target)  # backward pass  loss.backward()  # parameter update  optimizer.step()  # track loss  train\_losses.append(float(loss.item()))  train\_losses = np.array(train\_losses)  return train\_losses  def validate():  model.eval()  valid\_losses = []  y\_true, y\_prob = [], []  with torch.no\_grad():  for batch, target in valid\_loader:  # move data to the device  batch = batch.to(device)  target = target.to(device)  # make predictions  predictions = model(batch)  # calculate loss  loss = criterion(predictions, target)    # logits -> probabilities  torch.sigmoid\_(predictions)  # track losses and predictions  valid\_losses.append(float(loss.item()))  y\_true.extend(target.cpu().numpy())  y\_prob.extend(predictions.cpu().numpy())    y\_true = np.array(y\_true)  y\_prob = np.array(y\_prob)  y\_pred = y\_prob > 0.5  valid\_losses = np.array(valid\_losses)  # calculate validation accuracy from y\_true and y\_pred  fan\_accuracy = metrics.accuracy\_score(y\_true[:,0], y\_pred[:,0])  blotch\_accuracy = metrics.accuracy\_score(y\_true[:,1], y\_pred[:,1])  exact\_accuracy = np.all(y\_true == y\_pred, axis=1).mean()  # calculate the mean validation loss  valid\_loss = valid\_losses.mean()  return valid\_loss, fan\_accuracy, blotch\_accuracy, exact\_accuracy  def train(epochs, first\_epoch=1):  for epoch in range(first\_epoch, epochs+first\_epoch):  # train  train\_loss = train\_for\_epoch(optimizer)  # validation  valid\_loss, fan\_accuracy, blotch\_accuracy, both\_accuracy = validate()  print(f'[{epoch:02d}] train loss: {train\_loss.mean():0.04f} '  f'valid loss: {valid\_loss:0.04f} ',  f'fan acc: {fan\_accuracy:0.04f} ',  f'blotch acc: {blotch\_accuracy:0.04f} ',  f'both acc: {both\_accuracy:0.04f}'  )    # update learning curves  avg\_train\_losses.append(train\_loss.mean())  avg\_valid\_losses.append(valid\_loss)  valid\_accuracies.append((fan\_accuracy, blotch\_accuracy, both\_accuracy))    # save checkpoint  torch.save(model, f'/content/gdrive/MyDrive/ML/data/PlanetFour/checkpoints/baseline\_{epoch:03d}.pkl')  ## Constant classifier accuracy  Evaluate how accurate would a $f(x) = \text{"most common class"}$ classifier be?  def constant\_clf\_accuracy():  y\_true, y\_pred = [], []  with torch.no\_grad():  for \_, target in valid\_loader:  y\_true.extend(target.cpu().numpy())  y\_pred.extend(np.ones((target.shape[0], 2), dtype=np.float32))    y\_true = np.array(y\_true)  y\_pred = np.array(y\_pred)    # calculate validation accuracy from y\_true and y\_pred  f = metrics.accuracy\_score(y\_true[:,0], y\_pred[:,0])  b = metrics.accuracy\_score(y\_true[:,1], y\_pred[:,1])  t = np.all(y\_true == y\_pred, axis=1).mean()  print(f'fan: {f} blotch: {b} both: {t}')  constant\_clf\_accuracy()  ## Train the model  Call the ``train(n)`` function to train for ``n`` epochs.  train(5)  ### Plot loss and accuracy  def plot\_loss(avg\_train\_losses, avg\_valid\_losses):  validation\_losses = pd.DataFrame(avg\_valid\_losses)  training\_losses = pd.DataFrame(avg\_train\_losses)  plt.plot(validation\_losses)  plt.plot(training\_losses)  plt.ylabel("loss")  plt.xlabel("epoch")  plt.legend(["validation", "training"])  plt.show()  def plot\_accuracy(valid\_accuracies):  accuracies = pd.DataFrame(valid\_accuracies)  plt.plot(accuracies)  plt.ylabel("accuracy")  plt.xlabel("epoch")  plt.legend(["fan", "blotch", "both"])  plt.show()  plot\_loss(avg\_train\_losses, avg\_valid\_losses)  plot\_accuracy(valid\_accuracies) |