Human Activity Recognition using Random Forest and CNN/LSTM Hybrid models

Dafydd James Thomas

Abstract

This report outlines the rationale for the design (& subsequent training) of two Machine Learning models, built for the purpose of Human Activity Recognition (HAR) using the WISDM dataset. While the "Deep" model is trained directly on the raw accelerometer signal data, the "Shallow" model is developed from a set of handcrafted features.

CONTENTS

I. Introduction

Human Activity Recognition (HAR) tasks involve developing technologies that, with some input, predict/classify the corresponding activity. They exist in many forms (using inputs from wearable/ambient/object sensors, or using vision-based techniques, etc. [1]), & are useful for various purposes. An oft cited usage is to help learn/correct/improve the habits of individuals for health or lifestyle related purposes. The optimal inner-workings of such technologies remain of research interest, especially since the body of data & range of represented tasks is always expanding.

WISDM [2] is a collection of smartphone tri-axial accelerometer signals, in 8-second sequences, with labels specifying the corresponding task. In this report, I present two Machine Learning (ML) models that use the WISDM data to perform HAR tasks. The models differ by their associated input: one model is developed on the *Raw-Signal* data, taken directly from the smartphone accelerometers; the other uses *Handcrafted-Feature* data, which is extracted from the *Raw-Signal* data.

I found that, under various configurations, the best Random Forest model returned a score of 0.80559 on test data, whilst the CNN/LSTM Hybrid model returned a best score of 0.78401.

Here is a brief outline of the ensuing discussion. In section II, I discuss the *pre-processing* stage for handling the dataset - broadly, this covers handling any null or problematic samples, and addresses the normalization process for subsequent processing.

Section III is thereafter dedicated to fully discuss & justify the chosen classification models. Subsection III-A presents the case for using Random Forest as the classification method in the *Shallow* [1] model case. Equivalently, subsection III-B is dedicated to defending CNN/LSTM Hybrid as the choice of *Deep* model architecture.

Section IV then contains a lengthy discussion of each model's performance (IV-B for Random Forest and IV-C for CNN/LSTM Hybrid). Ultimately, this section serves as a record for the challenging task of improving the models under experimentation.

Finally, I conclude by summarizing the models and some reflective commentary in Section V. The primary theme of these remarks is hinged upon more appropriate planning/preparation, deeper consideration of alternative classifiers for the shallow model and a firmer appreciation for the domain-transfer problem.

II. Pre-Processing

A. Aspects of the Data

To begin, it's worth briefly describing what each dataset contains:

- *'id'*: The index of each sample
- *'user'*: 25 unique user IDs
- *'activity'*: The **Class Label** for each sample
- (*Handcrafted-Feature* data) *49 Numerical features*: Input
- (*Raw-Signal* data) *3 accelerometer signals (160-length arrays)*: Input

Of course, we take notice primarily of the Input & Class-labels for each sample, but it is worth mentioning that *user ID* is itself not unworthy of attention. Intuitively, *if* a model predicts perfectly, it should embody the User meta-information: *different users have different activity patterns*.

As an extreme example, input from *user_x* "Walking" is identical to *user_y* "Jogging" - to correctly predict both, the model must know something about these users. Abstracting away this information is important since then the model can still learn something about *unseen* users.

This deeper consideration is not accounted for with the Random Forest model, but is broadly discussed in Sections III-B and IV-C.

B. Data Cleaning

The first stage of dealing with any data is quality assurance: Are there problems within the data that inhibit downstream tasks? Often, this might be "mislabelled" data for a textual feature, or missing (NULL) / infinite / peculiar data for numeric features. Table I presents the relevant issues & how they were resolved. For this stage, data was passed in & out of `preprocessing.ipynb`.

$$\text{mag\_energy} = \sum_{t=1}^{N} (x_t^2 + y_t^2 + z_t^2)^2 \tag{1}$$

C. Standardization

1) Raw-Signal data

Standardization efforts are discussed more in detail throughout Section IV-C.

[1] I use the term "Shallow" to refer to the *Handcrafted-Feature* based model - this is common practice & is helpful to distinguish between models.

Issue	Description	Solution
NULL signal-values	In ID `s_1994`, the z array in the *Raw-Signal* data contains one missing value. This propagates into *xz_corr* and *yz_corr* in the *Handcrafted-Feature* data.	Since this is a singular/exceptional case, I **Impute** from the *mean of neighbouring values*. Human acceleration is typically smooth, especially when measured 20Hz (50ms intervals).
Empty signal-arrays	Many samples in the *Raw-Signal* data have arrays full of zeros. These rows are mostly labelled "Jogging" ($\frac{68}{70}$ rows, other 2 rows are "Upstairs"), suggesting *default* labels despite no actual movement. Corresponding rows in *Handcrafted-Feature* are full of NAs (by virtue of deriving from 0's).	**Exclude** these from training and testing (They are misleading anomalies). At prediction stage, Add rule-based logic (if signal-empty, then "Jogging") - Reflecting the default-label behaviour observed in training-data.
inf Feature values	In two rows of the *Handcrafted-Feature* test set, *mag_energy* is infinite, likely due to a pipeline error. The corresponding *Raw-Signal* rows are unaffected.	**Recalculate** *mag_energy* (see Eq. 1) from the *Raw-Signal* values. Sanity checks show calculated values are within 0.12% of original values.

TABLE I: Summary of Data Cleaning Stage.

2) Handcrafted-Feature data

The features that form the *Handcrafted-Feature* data include. E.g., Pearson Correlations ($[-1, 1]$) and FFT peak-frequencies ($[0, 80]$)[2]. Without normalization[3], classification will attend more to higher-variance features - this bias is harmful to proper prediction mechanisms. This is achieved using the via sklearn's `StandardScaler`, equivalent to Eq. 2, transforms features to scales with a mean of 0 & standard deviation of 1. With this, intra-feature variation is preserved whilst equalizing/normalizing inter-feature scales.

$$F_{\text{Norm}} = \frac{F - \mu_F}{\sigma_F} \tag{2}$$

D. Class Distribution

An additional consideration is: How balanced are class-labels in the dataset? Much attention is dedicated to this property during Section IV, but at the stage of exploration, I was clear to note that the training data exhibits a *strong* imbalance in class-distributions, as is demonstrated in Fig. 1.

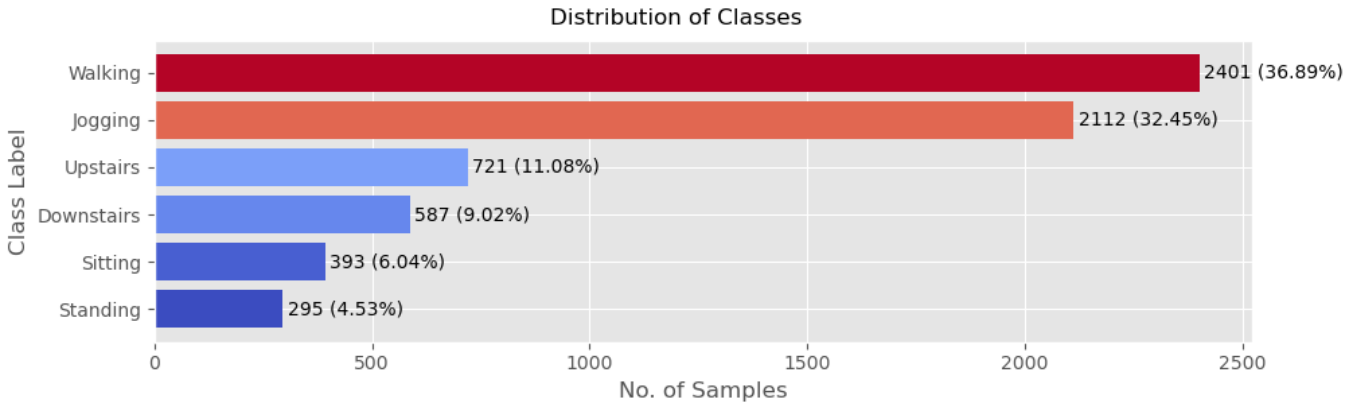

Fig. 1: A summary of Class Distributions in the Training Data

III. Methods

What follows is an explanation of my chosen classifiers. As motivation, I based my decision on the models performance(s) under scrutiny (as detailed in Section IV) according to their *capacity to generalize* (i.e. the **F1-Score at submission**). As such, other criteria (e.g., Compute or Time requirements) are not considered in equal standing; instead, I shall only briefly comment on these where notable.

A. Shallow Model

As an initial (baseline) model, I chose the **Random-Forest** classifier as developed by [4], which is often a good choice due to its strong foundation & interpretability. Moreover, its basic generalization capacity is supported by it being an *ensemble machine-learning method*.

[2]A Fast Fourier Transform (FFT) illustrates the amplitude of $\sin(x)$ frequencies present within a signal. A minimum frequency is for the wave that doesn't oscillate (i.e. $f = 0$). FFT's have an associated "Nyquist" frequency, f_s, which is determined by the sampling-rate (160 in our case): a total of $\frac{f_s}{2}$ (80) frequency bins (i.e. from 0 to 79) are detectable by an FFT. This is because the fastest identifiable sine-wave is one that successively peaks/troughs at every measurement (If the sigmoid wave is faster, it begins looking like it is slower due to the limitations in measurement)

[3]"Normalization" and "Standardization" are used interchangeably

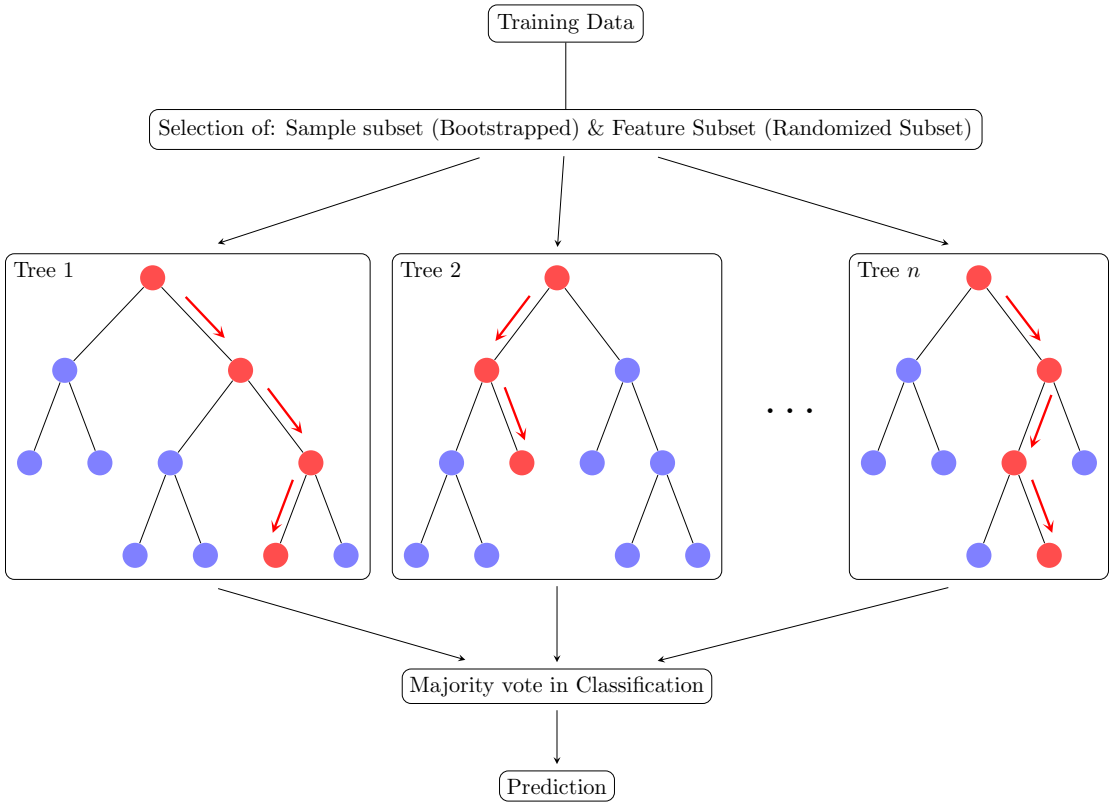

Fig. 2: A diagram of the Random Forest Model (Adapted from [3]'s version, originally published under CC BY-SA 4.0.)

An ensemble method is an ML approach that uses *a collection* of independent learners, and *aggregates* their classifications. They're considered to be robust because of their strong disposition to produce *unbiased* outputs. As such, ensemble methods are characterized by three attributes (Learning, Diversifying & Aggregating) which, for a Random Forest, are as follows:

1) ***Learning****: Decision Trees*

A Decision Tree (DT) is a method by which *rules* are found that divide samples into groups, iteratively, with the end-goal of *decreasing class impurity* in the groups. DTs are strongly rooted in information theory, because their guiding principle is to maximally reduce *entropy* (i.e. disorder/mix-of-classes) in the sample-set at each step. This process is guided by loss criteria such as *entropy-difference* or *Gini impurity*, which dictate the optimal split at each stage in the tree.

To illustrate the concept, consider a game in which one player selects an object and another has 20 yes/no questions to identify it. The optimal strategy is to ask questions that divide the remaining possibilities roughly in half. In this sense, both outcomes are equally informative. Any other choice is suboptimal *in the long run* - a question that eliminates 99% of possibilities if "yes" is wasteful if "No" (eliminates *only* 1%).

The DT decision-points ultimately resemble these questions, except they are made high-dimensional with the numeric data being continuous (e.g., *is* x *within* (a, b) *or* (c, d) *or* (e, f)*?*), and when there's a high number of features. I.e. with more features, there is a higher *branching factor* - this can be controlled for, but it is important to note that this fact limits the realistic possibilities of having *very deep* trees.

Moreover, they face a limitation in that *they draw linear decision boundaries* in the feature-space. There's not much scope for DTs to incorporate complex feature-interactions, or non-linear relationships within features.

2) ***Diversifying****: Bootstrapping & Randomized Feature Subsets*

So, a random forest makes use of many DTs. However, it is also necessary that each tree is *exposed to different* elements. They are algorithmically designed to achieve the same, so their decisions will be identical given the same data.

RFs make use of two methods to *improve diversity* among its learners. First, a DT only gets to see a *random subset* of features. This keeps individual trees efficient (lower dimensions) & brings a richness to the overall appreciation of each feature.

The second aspect of diversification is called **bootstrapping**: Each DT is trained on a set of samples taken from the dataset *with replacement*. For example, if my dataset is "A","B","C", then two valid *bootstrapped* sample sets are "A","A","C" and "A","B","A". The key idea is that with a larger set of bootstraps, though each specific bootstrap is unlikely to resemble the training-data, *the limiting behaviour of them as a group does*. Though not strictly an equivalence, the Central Limit Theorem (CLT) is an analogous concept [5].

So the bootstrap, alongside random feature subsets, are powerful & *stable* methods of increasing how diverse the DTs are.

*3) **Aggregating**: Majority Voting*

Now, every DT has drawn boundaries differently. For unseen samples, these trees will *necessarily* have different classifications. The RF model chooses the most popular (the *majority's* choice) of classifications, mimicking democratic ideals among DTs! This makes RFs robust in the face of outliers, but over-participation (i.e. too many DTs), this majority-voting technique is highly granular & specific to the DTs learned structures. So, as ever, there is a balance to be struck where the RF is complex enough to improve diversity & generalization, but not *too complex* that it overfits to the training-data.

B. CNN/LSTM Hybrid (Deep Learning)

For the end-to-end model, I was inspired to emulate the *DeepConvLSTM* model that Ordóñez and Roggen (2016) pioneer [6]. Their seminal paper proposes a *Hybrid* model, composed of several **Convolutional** and (subsequent) **Recurrent** (specifically, **LSTM**) Neural Network layers.

1) Convolutional Neural Network (CNN)

Multi-Layer Perceptrons (MLPs), equipped with backpropagation & the principles of gradient descent, are highly adaptive and powerful learners. They are, however, ill-suited to process a (3×160)-signal input, because by treating each channel/time-step as an input node, we're suggesting an *independence* between each input. To apply the principles of backpropagation & gradient descent to sequential, or image-data, Convolutional Neural Networks (CNNs) are powerful because they behave as *natural feature-extractors*.

A CNN is composed as follows

1) **Convolution**: depicted in Fig. 3, a convolutional layer uses *filters/kernels* ($\mathbf{K}$) to transform a k-length (Kernel size) window of 1D ($\times C$ channels) data into one value. These filters behave as extractors of k-timestep sized feature (across each channel). It helps to imagine the process as a dot-product operation: if $\mathbf{x}$ (input) highly resembles the filter $\mathbf{K}$, then the corresponding output (convolution) is high, because the input strongly matches the pattern of the filter. **Learnable Parameters**: Filters.
2) (Optional) **Batch Normalization**: standardizing outputs *within* a deep-model's layers [7]. Consider that a model updates its parameters per batch-run: the desired effect being that the next output loss is lower. However, by changing weights in layer l the *outputs* are different, which is disruptive of layer $(l+1)$'s learning trajectory[4]. Batch-Norm *stabilizes* outputs from a layer so that output distributions do not vary so wildly, speeding up the learning process. Operations using **learned parameters** γ_l and β_l(scale & shift) then re-contextualise the data for the *next layer's input*.
3) **Activation**: As with an MLP, non-linear patterns are only learned if outputs (convolutions) are passed through an *activation* function (e.g. ReLU, Sigmoid or Tanh).
4) (Optional) **Pooling**: Pooling is a method of *reducing the input-space resolution*. (i.e. "Downsampling"). This reduces the time-dimension whilst preserving the more valuable temporal information. In CNNs, the *maximum* value of k-neighbors (kernel-size) at s stride-lengths is known as *max-pooling*. E.g., $k = 2$ and $s = 2$, the temporal sequence is halved, where $\text{NewSequence}_i = \max(\text{OldSequence}_{[(si):(si+k)]})$.
5) **Dense Layer**: Using the obtained feature-maps as input-nodes, outputs can be produced via one or more dense/fully-connected layers as in regular MLPs, also with activation functions. **Learnable parameters** are the weights & biases as in MLPs.

$$\mathbf{I} * \mathbf{K} = \mathbf{W}^i \mathbf{x} + \mathbf{b} \tag{3}$$

where $\mathbf{W}^i \in \mathbb{R}^{(k \times C)}$ is the i^{th} filter for a kernel-size, k, and number of channels, C; $b \in \mathbb{R}$ is scalar bias term.

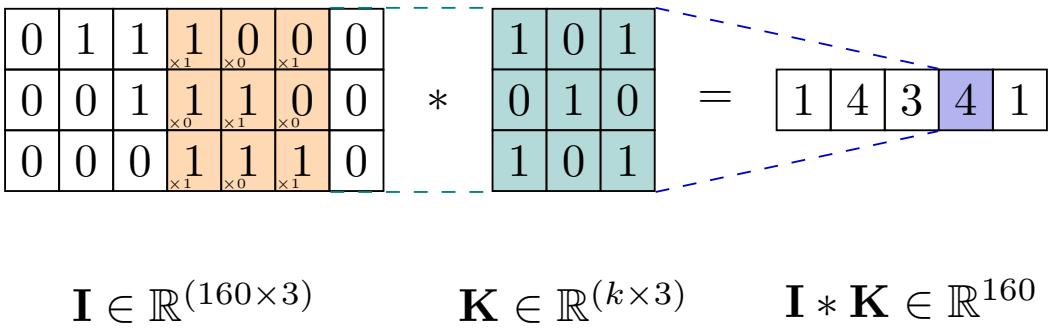

Fig. 3: Diagram of the Random Forest Model (Adapted from [3])

[4]By the fact that $\text{Out}_l = \text{Input}_{l+1}$

2) Recurrent Neural Network (RNN) & Long-Short Term Memory (LSTM) Models

The previous CNN layer succeeds in isolating important localized features, and while they can downsample the sequence of features to be more compact. However, there remains information *within* the temporal/sequential aspect that is not captured solely by CNNs. For this purpose, a further specialized deep-learning architecture is necessary to identify *global* patterns, namely *Recurrent Neural Networks* (RNNs). For HAR tasks, these are useful because it's important for the classifier to identify salient local patterns, as well as how these patterns evolve over the course of the sequence.

The process that delineate RNNs is fairly straightforward:

1) Using Inputs as a sequence ($\mathbf{x}_t \in \mathbb{R}^f$ where $t \in \{1, \ldots, \tau\}$ for a sequence of length τ with f of features per input)
2) $\mathbf{h}_0$ (Initial Hidden state) is specified
3) For each t, a *new* hidden state $\mathbf{h}_t$is calculated as a function of $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$, with separate (learnable) weight & bias parameters (See eq. 4)
4) The RNN processes a sequence *sequentially*, and outputs a new sequence of *hidden states*.

$$\mathbf{h}_t = \sigma(\mathbf{W}_x \mathbf{x}_t + \mathbf{W}_h \mathbf{h}_{t-1}) \tag{4}$$

As an example of *combining* CNNs and RNNs, consider that for a *sequence of words* (i.e. a sentence), the CNN learns features *within* local instances (i.e. word/phrase-level) in the sequence. With these features, the RNN learns how they evolve *between* local instances: The RNN outputs a new sequence, which is a representation of the sequential-evolution of these features *up until that element*. I.e., the ith element of the RNN output sequence is a representation of the sequence $x_{1:i}$ of inputs. Usually, however, it is the *final* element that matters, since this captures the *whole*-sentence.

However, because RNNs suffer the so-called *vanishing-gradient* problem[5], a more powerful & useful variant is the **Long-Short Term Memory** units (LSTMs) - long-term information is preserved & updated in accordance with new/different information. In a simplified sense, they are constructed so that long-term information can propagate further, and information is allowed to be ignored/*forgotten* in accordance with what information it provides to augment the sequence.

Of course, this "learning of context" can be done not only forwards through the sequence, but backwards too; this is known as a *Bidirectional* RNN or LSTM. Typically, this involves generating two separate hidden states for each input: one (h)is the hidden state of the forward-moving sequence, and the other (g) for the backward-moving sequence. Intuitively, this is like learning a sentence's meaning by reading it backwards instead of forwards. For many instances of sequence data, prediction is only meaningful in one (forward) direction.

However, as in this HAR scenario, we treat the signal-inputs as an atomic unit, wherein context relating to the task can be learned either way. E.g. We might expect that the similarity between forward-context and backward-context to be indicative of periodic behaviour, which is typical of "Walking"/"Jogging". The results of using BiLSTMs or regular LSTMs is discussed in Section IV-C.

IV. Experiments & Results

A. Experimental Process

NOTE Briefly, I feel it's worth noting how my workflow process evolved. For the *shallow* model development, the model-building began haphazardly[6], but gradually, I developed the process as follows:

1) **Build Model$_i$ (Parent)**: Analyse its content/performance.
2) **Extend into j^{th} Model$_i^j$ (Child)**: Experiment with parameters
3) **Find best of k children, Model$_i^*$**: **Save** the configuration & **upload** for test-performance, Score$_i^*$
4) **Compare Test Scores**: If Score$_i^*$ > Score$_i$, return to step 1. Else, proceed to *new* iteration with Model$_{i+1}$ as the Parent.

I utilized `sklearn`'s `Pipeline` class to create structured objects of each model's pipeline, and wrote customized functions to write/append/read/overwrite these configurations in a `model_config.json` file. This allowed for rigorous & *reproducible* documentation of models. In future, to limit the time/effort/errors committed in this process, I will look to use a more robust *version control* system for different models.

B. Shallow Model

3 categories of experiments were considered,

1) RF Classifier Hyperparameters
2) Feaure-Engineering (PCA & Feature-removal)
3) Under/Over-Sampling

[5]From the calculated loss of a model's prediction, backpropagation finds the gradient which dictates how parameters in the network should change. In some architectures, the prediction relies heavily on later layers - these layers will inherit much larger gradients, and the gradients for earlier layers diminish to 0. This inhibits these earlier layers, and hence the whole model, from learning as quickly/effectively [8]

[6]Due to this evolution of process, an earlier model ("model_1") configuration was unfortunately lost, so reproduction of its predictions are no longer feasible; the best approximation replaces its content in my `model_config.json` file.

For each category, experiments were cross-validated (CV) using `StratifiedKFold`[7] and various CV methods.

1) RF Classifier Hyperparameters

Recalling that earlier in Section III-A, there are some essential properties that govern how a Random Forest model functions:

- Number of Trees
- Depth of Trees
- Number of Features per Tree
- Criterion for impurity (loss-function) calculations
- Class Weighting in loss-function calculations

Each aspect was tested in turn. Specifically, earlier models used `RandomizedSearchCV` to find optimal parameters. Each parameter was then tweaked alongside others with `GridSearchCV`, for greater *rigour*, and plotted for exploration.

For example, comparing the effects of `n_esimators` and `max_depth` (as in Fig 4), At (600 estimators, 20 depth), the optimal score is reached. Generally, Score is increasing in n_estimators; the same can be said for max_depth, but beyond 15, the score changes very little on this dimension. Additionally, both attributes are (broadly) increasing in fit-time, so minimizing their size is essential for an efficient model.

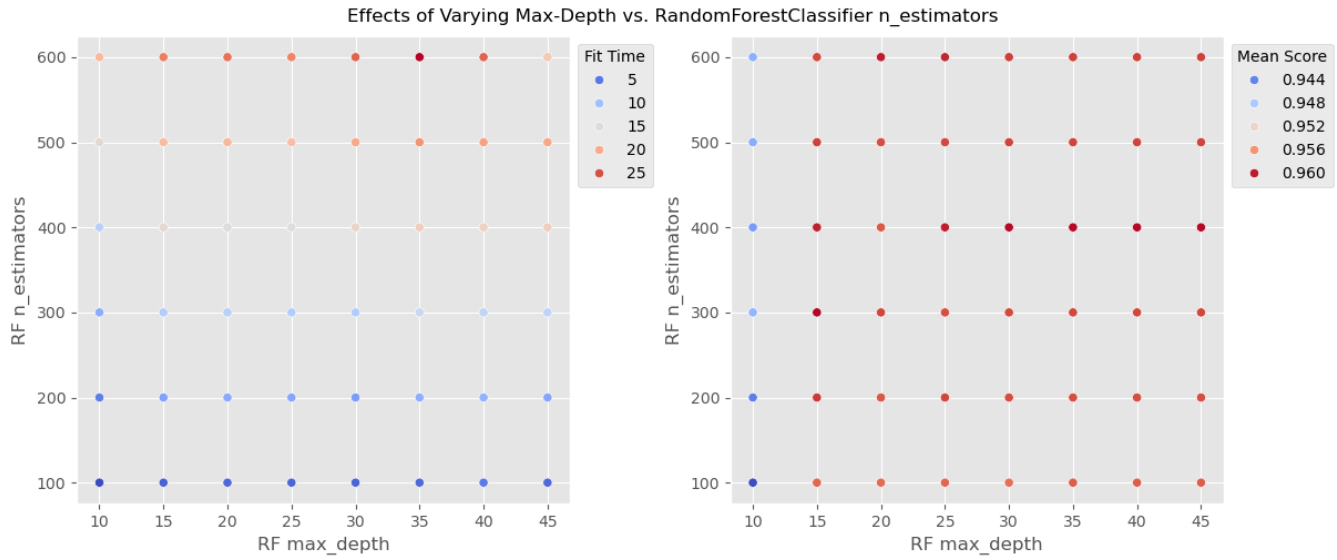

Fig. 4: Effects of changing max_depth & n_estimators

Table II summarizes the results of each hyperparameter experiment & justifies the resulting optimal parameter set:

Parameter	Optimal Setting	Comments
n_estimators	600	Struck the best balance between Lower & Larger values on Test performance. Consistently, 600 Trees outperforms fewer no. of Trees alongside other experiments
max_depth	20	Lower max_depth is faster but underfits; opposite trend for higher max_depth values
class_weight	'balanced'	Controlling for imbalance in training-data; far outperforms *None* weighting
criterion	'entropy'	Small difference
max_features	'sqrt' (DEFAULT)	Optimally each tree is designated no more than 7 features ($\sqrt{N_{\text{features}}}$) to prevent excessive training-time & overfitting in individual trees

TABLE II: Summary of RF-Hyperparameter Experiment Results

2) Feature Engineering

A topic I forwent discussing in Section II was *Feature Engineering*, which usually is considered a stage of preprocessing. However, for the Random Forest model, I was keen to see how feature-engineering affects performance (if at all).

A presumed method of feature-engineering uses `StandardScaler` to **Normalize** the scales/variance of each feature. In this sense, a prediction is not predicated on a feature's inherent value, but instead how it differs from other instances of that feature. This is strongly required to prevent a "large-scale" feature from overly dominating the classifier behaviour.

[7]For a multi-class dataset, this method is useful for an imbalanced class distribution. Each of the *k folds* are created with a similar distribution to the underlying training data, keeping this factor constant regardless of which *fold* is the validation set.

Methodology	Description	Dropped Features	Results
1. Threshold Importance value	If we have $49(\approx 50)$ features, then a dataset of *equally important features* has `feature_importances_` values at 0.02. Anything below is *less informative than average* so should be excluded.	19 features: `[x_mean, x_median, x_skew, x_kurtosis, x_mad, y_skew, z_mean, z_max, z_median, z_skew, x_fft_std, x_fft_max, y_fft_peak_freq, z_fft_std, z_fft_peak_freq, xz_corr, yz_corr, x_peak_dist_mean, z_peak_dist_mean]`	*Decreased* general performance: $0.80256 \rightarrow 0.79644$ (-0.612%)
2. 95% Cumulative Importance	Ordering each feature by importance, the set that cumulatively sums to 95% of importance should be kept; the remaining features should be excluded.	9 Features: `[z_fft_std, x_kurtosis, x_median, z_max, xz_corr, x_skew, y_skew, x_mean, x_fft_std]`	*Incerased* general performance: $0.80256 \rightarrow 0.80559$ $(+0.303\%)$

TABLE III: Summary of Feature-Exclusion methods

Another common method of reducing dimensionality before fitting an ML model is using **Principal Component Analysis** (PCA). This is a method of *changing basis* of the underlying features of the training-data, so that n-features are mapped onto the n-largest orthogonal eigenvectors of the $COV(X)$ matrix[8] (Each associated with its own eigenvalue, λ).

Though a bit of a mouthful, PCA returns linear-combinations of features as *independent components*, where each component explains some proportion of the data's variability. To reduce dimensionality without losing too much information, we can then drop the components with lowest eigenvalues (the lowest variability components).

Various settings of `n_components` (how many dimensions to keep) were tested, but optimally, this number is 0.

The last version of feature-engineering that I considered was dropping features *a priori* - i.e. removing features from the training/testing process. The rationale mirrors that which concerns PCA: Can we streamline the model, by removing unnecessary inputs? These inputs are those that have little-to-no variance, and not *meaningful* variance at that. In this sense, they're not *predictively powerful*, but *noisy*, serving only to slow down training & increase the risk of overfitting.

Table III summarizes the investigation of direct Feature Exclusion.

I based these decisions on information from `feature_importances_`[9], an attribute of `sklearn`'s `RandomForestClassifier` class. In Fig. 5 I've included an illustration of these values.

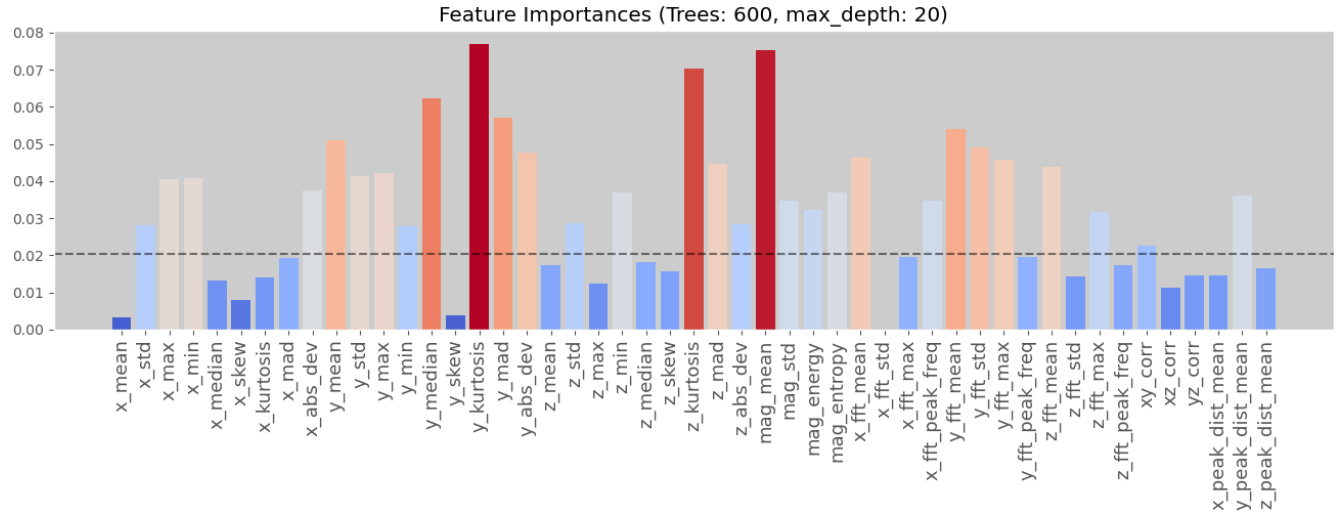

Fig. 5: A summary of Class Distributions in the Training Data

In fact, I conducted some further investigation (see Fig 6):

- *Cross-validation* performance is largely indifferent when excluding between 0 and 30 features [**Red** in Fig 6][10]

[8]If certain variables have drastically higher variance, $CORR(X)$ should be used instead. However, this concern is previously abated by **Normalization**.
[9]These *Feature Importance* values are defined by `sklearn` as the "mean decrease in impurity" as a result of using the respective features in DTs.
[10]Here it's worth noting the stark difference between *CV* and *Test* results; this discrepancy is common throughout the project.

- *Generality* performance (i.e. with Submissions on Test data) performance has an optimal level between 10 and 20 (large drop in performance at 25 & 30 excluded features) [**Blue** in Fig 6]
- The discussed methods in table III have an optimal level of 9 features.[**Green** in Fig 6]

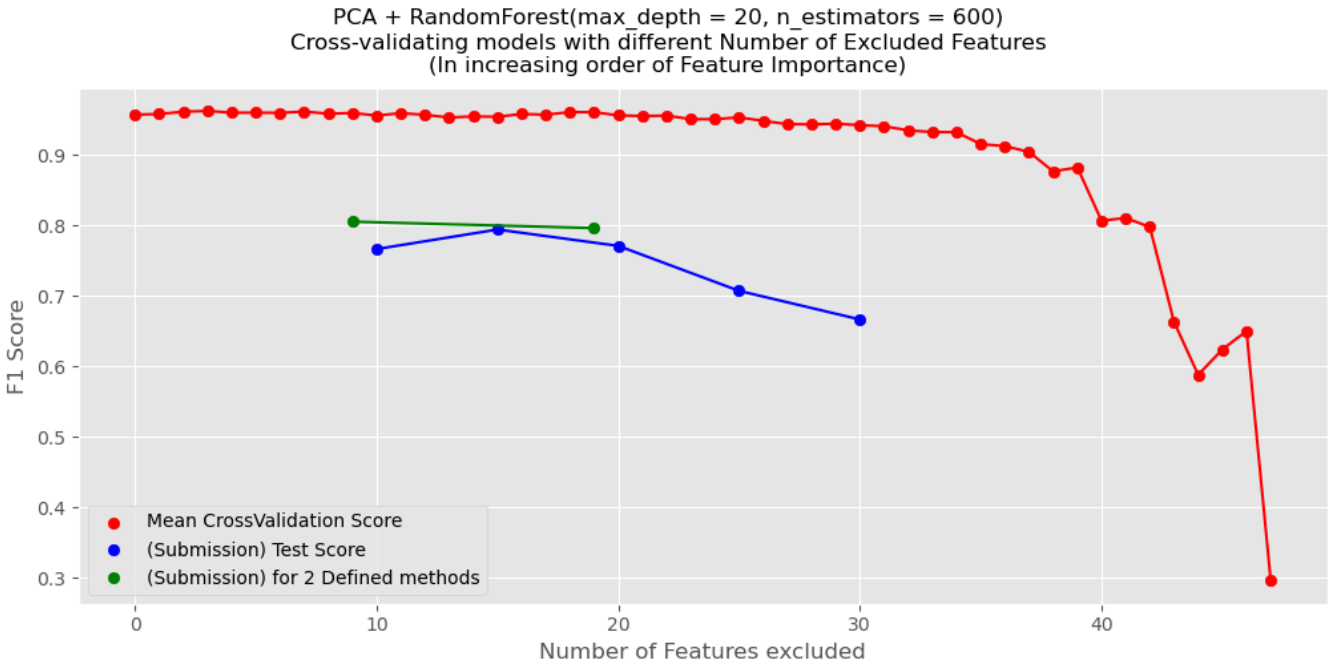

Fig. 6: A summary of Class Distributions in the Training Data

3) Under/Oversampling

A concern that lies outside the typical input (Feature-engineering), or processing (model hyperparameters) considerations is *class-representation in the training dataset*. Fig 1 shows the extent of this E.g. if in training, 90% samples are of class A, predictions will be biased in favour of A. In this scenario, this looks like a model predicting ”Walking” or ”Jogging” more often, by virtue of learning that these activities occur more often.

Our desire is for a model that makes predictions based on the *underlying nature* of an activity. To counteract the imbalance, we can use *Under-* or *Over-sampling* techniques to artificially balance the class distributions.

I tested 2 methods:

1) *Undersampling majority classes* using `RandomUnderSampler`
2) *Oversampling from minority classes* using `SMOTE` [9][11]

It was found however that there is no advantage to these methods as compared to using the argument (`class_weight='balanced'`)[12] in the `RandomForestClassifier` constructor, which is an alternative approach to resolving the class imbalance.

C. Deep Model

Given the objective of emulating that model which is developed in [6], the experiments which I chose to pursue here are fairly concentrated around the 4× CNN + 2× LSTM layer architecture.

For training, I developed a documentation/log of the models using `wandb`, and broadly structured the process to find:

1) Training Hyperparameters (Learning Rate, Batch Size, etc.)
2) Layer Configurations & Hyperparameters
3) User meta-data contextualizing methods
    - User Embeddings
    - User-Variable Normalization

Examples of the `wandb` dashboards (See Fig 7),

[11]In actuality, the authors of [9] demonstrate that LoRAS (*Localized Randomized Affine Shadow sampling*) outperforms SMOTE in most instances. However, SMOTE is used for its ease-of-integration with `sklearn`

[12]This method operates by adjusting the weight given to misclassification of each class during training. I.e. 'balanced' penalizes models for misclassifying *minority* classes more than for *majority* classes. [10]

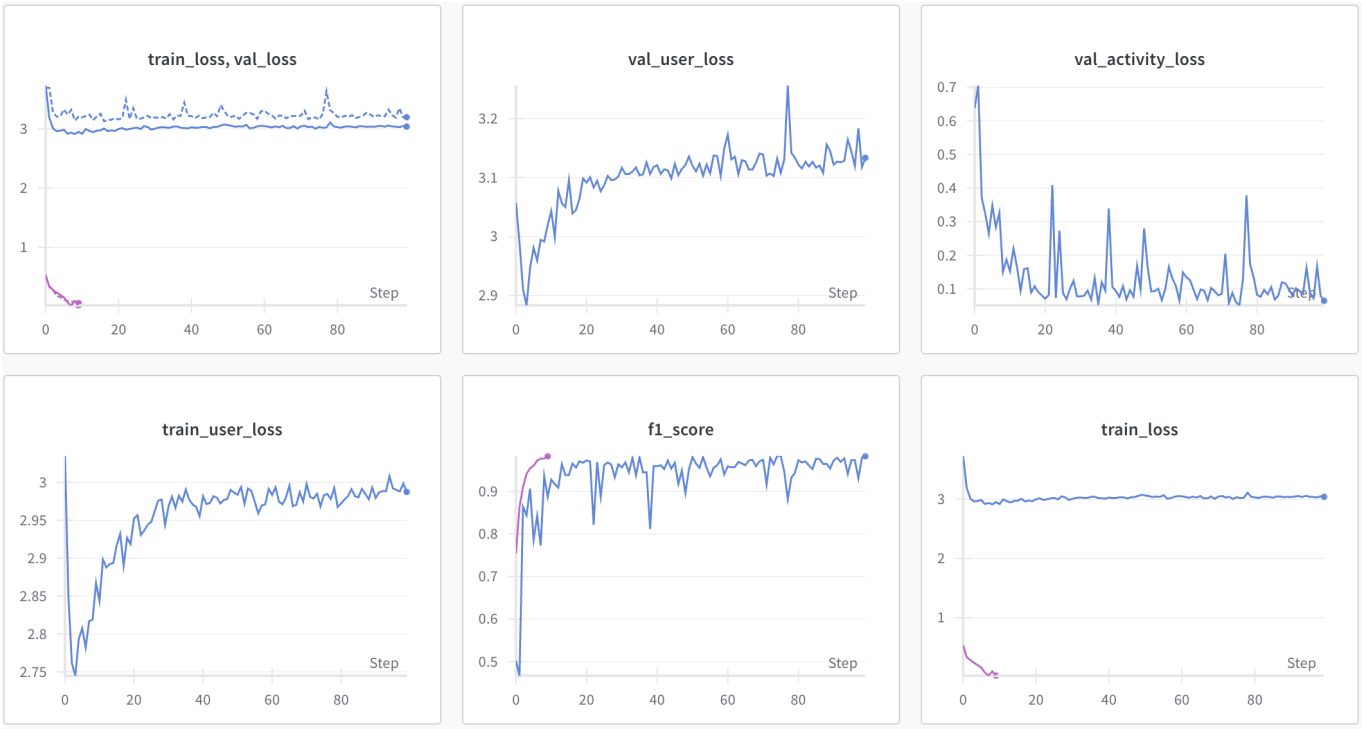

Fig. 7: A demonstration of the WANDB dashboard: In Blue is the Best result from Section IV-C4; purple is the 1-CNN-1-LSTM that is best overall (10-epoch convergence - discussed in Section IV-C2)

1) High-level problems

[11] discuss many of the issues that I subsequently faced during the process of building this deep model.

A recurring problem in resulting test-predictions was the *consistent* over-representation (or confused classification) of minority classes. This is highly typical behaviour of a model that *is overconfident* in predicting majority classes: If the model learns the "Jogging" samples in the training-set *really closely*, it is likely to scrutinize test-samples much more, and thereby predict "Jogging" much less regularly than it should. Conversely, the model is more uncertain about under-represented classes, and when a test-case is not so closely matching the majority classes, it will attempt an edge-case classification.

In part, this was because of improper input organization: *stratifying* the train/validation splits proved vital in addressing this problem, as well as thereafter using a *weighted* loss function[13].

Adding to this, I used the Test-predictions' class distribution as a *heuristic* for how well my model performed, since the best-performing Random Forest predictions were fairly similar to the training-dataset distribution. What was *consistently* observed was that, under different setups, "Walking" and "Jogging" were specifically being *misclassified* as "Upstairs" or "Downstairs". A major aspect of correcting this came down to using L2 Regularization to punish extreme weights, which is provably a strong method of limiting the capacity of the model to overfit for certain classes.

Mostly, however, this happened because of my failure in accounting for meta-data (i.e. Users). The process of this is discussed in further detail within Section IV-C4.

2) Baseline

As a baseline model, one CNN layer & one LSTM layer were utilized, with a final fully-connected layer to produce outputs. This model, with a training-time of ≈ 30 seconds and F1-submission score of 0.78401, performs quite well for the baseline. With a max-pooling layer, this architecture effectively:

1) Finds 64 features for each 5-timestep segment ($\mathbf{x} \in \mathbb{R}^{160\times64}$)
2) Takes the max of 2 successive feature-instances, compressing the sequence ($\mathbf{x} \in \mathbb{R}^{80\times64}$)
3) Models the temporal dependencies with an `LSTM`
4) *Final output* of the LSTM is taken & passed to a `Linear` dense layer to produce 6 outputs
5) The index of the highest value output is the predicted class ($\max$(DenseLayer) at 0th node $\rightarrow$ "Downstairs" for example)

The specific hyperparameters that informed this model are found in Table IV, largely inspired by those in [6].

[13]The loss function penalizes parameters by more/less, dependent on which class it failed to predict. Minority classes have higher weights, so the model is penalized more for incorrectly classifying minority classes - the tendency otherwise would be to underfit on these classes, and overfit to classes that are seen more often & are hence learned more deeply (majority classes).

Parameter	Value	Comments
Learning Rate, λ	0.001	As recommended by [6]; fast convergence achieved, without loss of generality, within roughly 10 epochs
Batch Size	32	[6] use batches of 100; 32 batch-size strikes a balance of efficient training with the relatively smaller size dataset
Train/Validation Split	80/20	Found to deliver more representable subsets of the population-level class-distributions, whilst maintaining a healthy portion of training-data to inform updates
Number of Epochs	$\approx$10	Tested up to 20 epochs, but Training-loss & performance-metrics typically tend to converge much sooner at around 10; beyond this, we risk overfitting to the training data

TABLE IV: Summary of RF-Hyperparameter Experiment Results

3) Architecture alternatives

The strong performance of the baseline architecture initially suggested many plausible directions for experimentation. However, **without incorporating user meta-data, no experimentation improved test performance over the baseline**.[14]

Given the vast design space, I focused on aspects where no prior intuition clearly favoured one choice:

- Number of channels (features) in CNN and LSTM layers
- Number and directionality of LSTM layers
- Kernel size, stride, and CNN stacking
- Pooling: does it help?
- Dropout: useful for generalization?
- LSTM output: final timestep vs. pooled sequence output

For simplicity, I varied channel counts uniformly across all CNN layers; future work could test varying them across layers. The results followed expected patterns: increasing complexity raised computational cost, with diminishing returns beyond certain configurations.

Key findings:

- **Stacked CNN layers** were beneficial only when each successive layer increased kernel size (e.g., $k = 3$ then $k = 5$), suggesting stacked CNNs are more effective when learning progressively broader patterns. Increasing channel count beyond 32 yielded diminishing returns.
- **Global-pooling LSTM outputs** (concatenating average- and max-pooled vectors across the sequence) significantly outperformed using only the final timestep. This is justifiable because:
  - We classify fixed 8-second sequences rather than real-time streams.
  - For short-duration activities (e.g., sitting/standing), pooling preserves signals that might otherwise be overshadowed by inactivity.
- **LSTM performed best with 2 layers**; dropout and bidirectionality had little effect.

Table V summarizes the baseline architecture, which remained the strongest without transfer learning:

Layer/Stage	Hyperparameters	Extra Arguments
1D Convolution	64-feature maps from 3 input-channels	$k = 5$, `padding` = 'same'
Activation	ReLU	–
Max-Pooling	$[k, s] = [2, 2]$	–
LSTM	1 layer of 64-size hidden inputs	`bidirectional` = False
Dense (Fully-Connected)	Input is non-pooled LSTM final value	Size of layer: $80 \rightarrow 6$

TABLE V: Summary of Baseline Model Architecture

4) Transfer Learning: Domain (User meta-data) Adaptation

Here I discuss an aspect of the problem that has been paid little consideration thus far: the *'user_id'* element. Let's imagine we're trying to learn what "Running" looks like: a common principle in scientific inquiry is *Ceteris Paribus* ("all else held equal"), which means that we'll learn best what "Running" looks like (in comparison to "Walking") in the context of a single *person* (e.g. Bob). However, this does not mean that we can distinguish between a different person (e.g. Alice) "Running" or "Walking". This is because our concept of "Running" is actually "Bob-Running".

I.e. those *features* that we associate with "Running" are unintentionally encoding some features of "Bob". The WISDM dataset poses this problem too: Users are *different* in the Training and Test sets.

What's the Solution? We teach the model to learn features from the data (as per usual) to classify activities. From these learned-features, we instruct the model to *also classify the user*. We therefore get two predictions (Activity & User).

[14]The following reports changes based on validation performance; while insightful, these are less meaningful without corresponding improvements on test performance, since generalization remains unimproved.

As with usual backpropagation, we find *gradients* associated with parameters that produced these outputs based on loss-calculations, to iteratively work backwards and adjust these parameters to improve predictions in the next pass.

However, we want the model to learn user-invariant features. This means that we want the model to have little/no ability to correctly identify a User from a given input. We therefore *reverse* the direction of learning according to loss from the user prediction! This is known as **Adversarial training** [12]: a Model is encouraged *away* from user-specific features, and instead is instructed to maximize classification results from the relevant (user-invariant) features that relate to Activity.

In a technical sense, this is achieved by applying loss-learnings of step $-\lambda$ (where λ is the learning-rate) to the feature-extractor part of the model. Moreover, we *schedule* this learning rate to more strongly learn adversarially over time using an epoch-dependent α value (not to be confused with α for L2 Regularization)

This is done so that in earlier epochs, the feature-extractor can learn unimpeded, and after some time (when the learned-representations are more stable) we apply adversarial pressure to discourage the model from using the extracted features to correctly identify users.

This alternative was found to produce seemingly more *honest* Validation scores, which showed promise. However, perhaps as a result of improper/insufficient experimentation with architectural components, the resultant models remained unfavourable, both in terms of training time & performance, as compared to the Random Forest model.

At best, a 3-CNN-1-BiLSTM model (trained for 100 epochs[15]) results in ≈ 0.75 test (generalization) score.

V. Conclusion

The immense scope for variation and alternative modelling choices in Deep Learning is endlessly fascinating, but provably, the simpler methods of shallow-learning (Random Forest) are just as (if not more) effective in providing accurate classifications for HAR tasks on the WISDM dataset.

In future, there is no shortage of alternative methods/experiments that I can imagine applying:

- Adversarial Training in the Random Forest model.
- Alternative model-components (e.g. RNNs, GRUs, Attention-Layers, etc.).
- Alternative methods of pre-processing (Augmenting/Engineering the handcrafted-features).

Both models perform relatively well, but the Random Forest model is far superior in both its performance & computational/time efficiency.

A. Reflection

This project presented some major opportunities to develop & practice my practical ML skills, as well as my ML-research skills. The major challenge I faced came in the form of proper planning/preparation for the process: I found that largely, the methodology would evolve as I incorporated new considerations along the way. With this experience in mind, I will endeavour to dedicate more time/consideration to devising a plan in the future. This will help with securing reproducible & rigorous patterns of development/experimentation.

As well as this, extending the planning-stage would allow for further considerations of relevant work, which would benefit the higher-level considerations that govern the experimental approach. I.e., as well as using [6] to support my decisions, I might have dedicated more time to considering the flaws/improvements to this model, in order to both avoid pitfalls & strengthen the narrative for my experimental choices.

References

[1] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu, "Deep learning for sensor-based activity recognition: A survey," *Pattern Recognition Letters*, vol. 119, pp. 3–11, Mar. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016786551830045X

[2] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, "Activity recognition using cell phone accelerometers," *SIGKDD Explor. Newsl.*, vol. 12, no. 2, pp. 74–82, Mar. 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/1964897.1964918

[3] J. Riebesell and S. Bringuier, "Collection of scientific diagrams," Aug. 2020. [Online]. Available: https://github.com/janosh/diagrams

[4] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[5] B. Efron, "Bootstrap Methods: Another Look at the Jackknife," *The Annals of Statistics*, vol. 7, no. 1, pp. 1–26, Jan. 1979, publisher: Institute of Mathematical Statistics. [Online]. Available: https://projecteuclid.org/journals/annals-of-statistics/volume-7/issue-1/Bootstrap-Methods-Another-Look-at-the-Jackknife/10.1214/aos/1176344552.full

[6] F. J. Ordóñez and D. Roggen, "Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition," *Sensors*, vol. 16, no. 1, p. 115, Jan. 2016, number: 1 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1424-8220/16/1/115

[7] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Mar. 2015, arXiv:1502.03167 [cs]. [Online]. Available: http://arxiv.org/abs/1502.03167

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[9] M. Hanif, R. Ahmad, S. Qazi, W. Ahmed, A. Kiani, and M. M. Alam, "Human Activity Recognition Using WISDM: Exploring Class Balancing and ML Techniques," in *2024 5th International Conference on Emerging Trends in Electrical, Electronic and Communications Engineering (ELECOM)*. Balaclava, Mauritius: IEEE, Nov. 2024, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10892156/

[10] "RandomForestClassifier." [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[11] K. Chen, D. Zhang, L. Yao, B. Guo, Z. Yu, and Y. Liu, "Deep Learning for Sensor-based Human Activity Recognition: Overview, Challenges and Opportunities," Jan. 2021, arXiv:2001.07416 [cs]. [Online]. Available: http://arxiv.org/abs/2001.07416

[12] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-Adversarial Training of Neural Networks," May 2016, arXiv:1505.07818 [stat]. [Online]. Available: http://arxiv.org/abs/1505.07818

[15] Generally it was found that higher num_epochs allowed for more precise convergence after implementing Adversarial training.