REPORT **Date** December 11, 2017 **Reference number**

Created by
Manuel Osvaldo Olguín

Final Assignment

DD3431 Machine Learning (PhD variant)

This final report for the DD3431 Machine Learning course describes the application of Linear Support Vector Machines to a multi-class, multi-output classification problem modeling the resource allocation strategies for basestations in a 5G environment. The classifier was adapted to the multi-class nature of the output.

1 Theoretical description of the problem

1.1 Description of the data

The data used for this study corresponds to samples of the resource allocation strategy for a communication system composed of exactly three basestations and up to a maximum of 20 user devices. The area of interest studied within this communication scenario is modeled as a map; more specifically, it represents an area of 2400 m^2 divided into a grid of 2 m×2 m cells. This grid is encoded as a matrix, with values between 0 and 2 indicating the occupancy status of each cell:

- 0 for empty cells,
- 1 for cells occupied by a user device,
- 2 for cells occupied by a basestation.

The occupancy data for each user can either be perfect or have an inaccuracy defined by a normal distribution with three possible standard deviations: 0.1, 0.25 or 0.4. Thus, the model had to be trained a total of four times, one for each variation of the positioning inaccuracy – we will call them "scenarios". The input data for each scenario corresponds then to the vectorized form of the corresponding matrix, i.e. a vector of 600* elements, each with a value $v_i \in \{0, 1, 2\}$.

On the other hand, the output data for the learning algorithm corresponds to encoded variables representing the basestation–user device association and the resource allocation information for each sample. The exact details of this encoding and the information carried within escape the scope of this report, but it is of importance to note that:

* $\frac{2400}{(2\times2)} = 600$

1. there are 9 of these output variables per input sample (3 for each basestation in the model);
2. these output variables are mutually independent;
3. they have discrete integer values that range between 0 and 12 288;
4. the order in which these values appear in the output is relevant (i.e. permutations of the same values correspond to different output classes!);
5. finally, these values can also be repeated across variables.

Items 4 and 5 make the problem different than a "regular" multi-label classification problem, in which only the distinct *sets* of labels matter [1]. 5 is in practice a consequence of 4 – the significance of the positions of the labels in the output means that repeated values don't have the same meaning since their positions in the output vector are different. Additional preprocessing was thus necessary to adapt the output to traditional multi-label classification methods; this is discussed in section 1.3.

In summary, the structure of an arbitrary sample looks like the following:

$$\underbrace{0, 1, 0, 0, 2, 0, \ldots 0, 0, 0, 0, 0, 1}_{\text{600 input features}}, \underbrace{4567, 23, \ldots, 1337}_{\text{9 output labels}} \tag{1}$$

In total, 72 000 samples in this format for each scenario were provided for the training of the model, with two thirds (48 000) of these used for fitting and the rest (24 000) used for validation purposes. The complete dataset was also used for 10-fold cross-validation, again for each scenario.

1.2 Adapting the data

1.3 Choice and adaptation of classifier

The dataset in question is part of ongoing research at the Department of Information Science and Engineering of the School of Electrical Engineering, and has already been modeled with great success using Random Forests and Neural Networks. The application of Support Vector Machines was then a natural step given this models' popularity in networking research literature; the specific application of the Linear Kernel for SVMs was a result of exploratory analysis with the dataset in the early stages of development, where initial experiments exposed the linearly separatable nature of the output variables.

Support Vector Machines are binary classifiers though, and thus additional modifications are required to adapt these classifiers to multiclass problems as the one in question. Specifically, the following techniques for adapting binary classifiers to multiclass applications were identified from literature [2, 3]:

One-vs-One Method: For a universe K of classes, with $\kappa = |K|$, this method constructs a maximum of $(\kappa(\kappa-1)/2)$ classifiers[†], each comparing a pair of classes from the training set. For prediction, all of these classifiers are applied to an unseen sample, whose final class corresponds to the class with the most "votes" after processing. In case of a tie, it selects the class with the highest total confidence, obtained by aggregating the confidence scores of each binary classifier.

One-vs-Rest Method: Constructs a maximum of κ classifiers, each comparing one class in the training set with the rest (i.e. each classifier determines if a sample belongs to a specific class or not). At prediction time, these classifiers are applied to the unseen sample and it is once again classified according to the majority vote.

Other multiclass adaptations of binary classifiers, like DAGSVM [4] and DDAG [5] were considered as well, but were ultimately judged to be overly complex for the problem at hand.

Note though that all of these modifications only transform a binary-class single-label classifier into a multi-classs single-label classifier, *i.e.* they don't adapt the classifier to output more than one output variable (label) per prediction. Thus, another modification to the problem was necessary, as changing the classifier was discarded from the beginning.

Two problem transformation strategies were evaluated: the *Binary Relevance* and the *Label Powerset* transformations [1]:

Binary Relevance: This problem transformation learns $\kappa = |K|$ binary classifiers – where K again is the universe of classes, *i.e.* one binary classifier for each class. Then, for an unseen sample i, it applies all κ classifiers to it and assigns to it the union of all labels returned.

Label Powerset: This transformation is based on treating each distinct set of labels in the training output as a separate *meta-label*. It thus trains *one* multi-class classifier on η input samples with $|\Lambda|$ possible output meta-labels, where Λ is the collection of unique label combinations present in the original training output. Since it basically converts the problem into a single multi-class classification scenario, this strategy only works well if the number of samples is much larger than $|\Lambda|$.

The final decision about which multi-class classifier and problem transformation to use was mostly based on time efficiency. The large dimensions of the problem meant that less time-efficient classification methods could take extremely long times to fit and validate, without proportional gains in accuracy. Thus, the in the end the following methods were selected for the assignment:

Multi-class classifier: The *One-vs-Rest* method was selected to adapt Support Vector Machines to the multi-class problem at hand, specifically because it only needs to train κ classifiers instead of $\kappa(\kappa-1)/2$ (this number could be very big).

[†]This number could be smaller, as only the classes actually present in the training ouput are used.

Problem transformation: It quickly became evident in the early exploratory stages of this assignment that the binary relevance method would be very unpractical. Given the large universe of labels of the problem ($K = [0, 12\,288] \Rightarrow |K| = 12\,289$) and the large number of samples available, this method would take large amounts of time to both fit and validate.

On the other hand, exploratory analysis of the input data revealed there were only about 800 distinct output label permutations for the 48 000 training samples for each scenario, which meant that the Label Powerset method was applicable to this problem.

This combination of methods meant that only *one* multi-class Support Vector Machine, comprised of about 800 binary SVMs (one for each *meta-label*), needed to be trained for each scenario. In contrast, the number of of classifiers that to be trained for each of the other combinations is:

- For the binary relevance transformation directly with binary SVM's (since this method works directly with binary classifiers), 12 289 classifiers need to be trained for each scenario.
- For *One-vs-One* multi-class SVM's using the Label Powerset transformation, approximately $800(800 - 1)/2 = 319\,600$ classifiers need to be trained.

Finally, traditional multi-label classification methods don't take into consideration the positions or the repetition of output labels – only the distinct *sets* of output labels are considered. Thus the problem had to be transformed in such a way that the positional information of each label in the output was encoded into the *meta-label* used by the Label Powerset transformation.

2 Implementation

2.1 Language and libraries used

The language chosen for this project was Python 3.6, given its extensive support for scientific programming, data analysis and machine learning in the form of libraries. In particular, the libraries **scikit-learn** (and its multilabel extension, **scikit-multilearn**), **scipy**, **matplotlib** and **numpy** were used for the implementation [6, 7, 8, 9, 10].

2.2 Choice of classifier

Based on the analysis detailed in section 1.3, **sklearn.svm.LinearSVC** was selected as the base classifier to be used for this problem, as it corresponds to an implementation of a multi-class Support Vector Machine using the *One-vs-Rest* method and a linear

kernel. Tests were also conducted with **sklearn.svm.SVC**, which implements a multi-class SVM using the *One-vs-One* method and a RBF-kernel, but its runtime proved to be cumbersomely large[‡].

The classifier was then extended to work on multilabel outputs through the **skmultilearn.problem_transform.LabelPowerSet** class, which implements the *Label Powerset* problem transformation as detailed in 1.2. Thus, the classifier treats every distinct label combination and/or permutation in the training data as a separate class when fitting and predicting.

2.3 Parsing and adapting the data

The data provided for building the model consisted of 72 000 samples for each of the four positional accuracy values detailed in the problem description (section 1.1 - for simplicity's sake we'll refer to them as four different *scenarios*), divided into two files each: 48 000 of the samples for training, and 24 000 for validation. The format of the input files was one sample per line: 600 integers with values in $\{0, 1, 2\}$ representing the input features, followed by 9 integers with values in $[0, 12288]$ representing the output variables (everything separated by commas, see example in listing 1).

```
0,0,0,...,0,2,0,...,3968,3841,528,8080,4209,6273,9201,9073,8592
0,0,0,...,0,2,0,...,80,2545,2417,6145,8176,4112,8443,11761,8736
0,0,0,...,0,2,0,...,3680,0,0,4112,8017,7408,10241,8272,11505
```

Listing 1: Abbreviated example of input samples.

This data was parsed using **numpy**'s **loadfromtxt()** function, and then split into separate arrays for the input and output data (labels). The input data was passed "as-is" to the classifier, but the output data required additional processing which will be detailed below.

As mentioned before, the output labels for each case were presented in the form of 9 consecutive integers, each ranging in value from 0 to 12 288. On the other hand, as per the API specifications[§] of **scikit-multilearn**, when using the **LabelPowerSet** problem transformation, the output labels passed to (and obtained from) the classifier need to be encoded into a *binary indicator matrix*.

Definition 1. A binary indicator matrix for a set of η samples, each with κ associated output labels out of a universe K of distinct possible labels, is a matrix A of dimensions $(\eta \times |K|)$, where

$$A_{i,j} = \begin{cases} 1 \text{ if label } j \in K \text{ is assigned to sample } i \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

Thus, A is a sparse matrix where each row has *at most* κ elements of value 1.

[‡]Not a fair comparison, but for the unoptimized SVC case the runtime was over 3 hours for fitting and predicting, whereas the final runtime for the optimized and parallelized LinearSVC is, on average, 110 seconds.

[§]http://scikit.ml/api/datasets.html#the-multi-label-data-representation

Note that binary indicator matrices *do not preserve information about label positions* in the output. Consider two sets of labels α and β, where β is a permutation of α – i.e. both have the same labels but in different order. Given the previous definition of a the binary indicator matrix, both sets would then have the same representation in the matrix, since each element of a row only indicates if the label is present in the output. They also don't handle repeated labels correctly – multiples of a label are all assigned to the same element of the matrix.

All of this poses a big problem for the model in question, where there can be repeated values in the output variables, and the order of which is relevant for the result. This meant that additional preprocessing of the data was necessary to encode the positional data of each output variable into the binary indicator matrix – this would additionally solve the repetition problem, since variables with equal values but different positions would have different encodings.

The following function was thus applied to each label in the output:

$$f_{\text{enc}}(v, i) = (i * |K|) + v \tag{3}$$

Where v is the value of the output variable (label), i corresponds to the 0-indexed position of the variable in the output vector and K is the universe of valid labels ($K = [0, 12288]$ in this particular case). In practice, the result is that the first variable of each sample is assigned a value between 0 and $|K| - 1$, the second variable a new value between $|K|$ and $2|K| - 1$, the third between $2|K|$ and $3|K| - 1$ and so on. This ensures that when encoding the output variables into a binary indicator matrix, variables retains the information about the position they occupied in the original vector – and since the new values are all distinct (since they depend on the individual position of each variable and no two variables share the same position) it also solves the problem of duplicate variables "disappearing" in the binary indicator matrix.

Finally, to revert the previous transformation (for instance, when getting the results from the classifier), one only needs to apply the following to each encoded value v':

$$v' = f_{\text{enc}}(v, i) \tag{4}$$

$$v = v' \bmod |K| \tag{5}$$

$$i = \frac{v' - v}{|K|} \tag{6}$$

These operations transform the previously encoded value into the "real" output value v and its position i in the final output vector.

2.4 Fitting and validating the model

Fitting and predicting on the test datasets and validating the results were performed in parallel for the four scenarios using a Python process pool (**multiprocessing.Pool**).

```
def encode_output(output):
    """
    Processes the given label set output matrix to the binary
        representation
    expected by scikit-multilearn
    :param output: Label set output matrix
    :return: Binary representation of input matrix
    """
    processed_output = dok_matrix((len(output),
                       n_labels * n_classes),
                       dtype=np.uint8)

    for i, row in enumerate(output):
        for j, label in enumerate(row):
            processed_output[i, (j * n_classes) + label] = 1

    return processed_output.tocsr()
```

Listing 2: Encoding function in Python. This function both transforms the given output label matrix using f_{enc} and encodes the result into a binary indicator matrix.

For each scenario, a classifier was constructed using the **LabelPowerSet** class with a **LinearSVC** with default parameters as the base classifier, which was then fitted on the training input data and the associated binary indicator matrix for the processed label output data using the **fit()** method of the classifier. Next, the model was validated on the training data with a custom validation function which takes advantage of the matrix representation of the results and of the fact that the position of each label matters in the final result. Let V be the binary indicator matrix for the labels for the test fraction of the dataset (i.e. the expected output), and V' the output generated by the classifier for the test input data, then the validation function f_v can be expressed as:

$$f_v = 1 - \frac{\text{nonzero}(V - V')}{2 \times \text{nonzero}(V)} \tag{7}$$

In words, this function first calculates the total number of non-zero elements in the element-by-element difference between both matrices; since both matrices have exactly the same amount of non-zero elements per row (by definition, thanks to the previously discussed encoding function), this amount will actually be double the real amount of discrepancies (since each "error" will appear twice in the difference matrix, once for the value in the validation matrix and once for the value in the output matrix). This value is then divided by two to get the correct amount of errors, before being divided by the total expected number of output values in the validation matrix to get the decimal representation of the error in the classification. Finally, this value is subtracted from 1 to get the total accuracy value of the model for the given training

```
def decode_output(encoded_output):
    """
    Decodes a binary indicator matrix into a 2-D vector
    of label vectors. Each row in the output corresponds
    to the label vector for a specific sample.
    :param encoded_output: Binary indicator matrix.
    :return: 2-D vector of label vectors.
    """
    n_zero = encoded_output.nonzero()
    decoded_output = np.empty(shape=(encoded_output.shape[0],
                      n_labels),
                      dtype=np.uint16)
    for row, col in zip(*n_zero):
        value = col % n_classes
        decoded_col = int(col - value) / n_classes
        decoded_output[int(row), int(decoded_col)] = value

    return decoded_output
```

Listing 3: Decoding function in Python, which takes a binary indicator matrix and extracts and decodes the encoded values within into the original data format.

data.

```
def accuracy_score(x1, x2):
    assert x1.shape == x2.shape
    assert x1.nnz == x2.nnz
    errors = (x1 - x2).nnz / 2.0
    return 1.0 - (errors / x1.nnz)
```

Listing 4: Accuracy score function implemented in Python

After validation of the results, the accuracy score values are printed to standard output for review. The predictions generated by the classifier for each scenario are "decoded" (using the function detailed in listing 3) back the to dataset format for output labels (i.e, vectors of nine integers between 0 and 12 288 each) and written to CSV-formatted text files for future use.

2.5 k-fold cross-validation

Although **scikit-learn** includes a function for k-fold cross-validation, it quickly became evident that an adapted, more optimized implementation was required. In particular, the built-in function uses the default accuracy score function, which doesn't take into account the position of the labels in the final output vector. The library's cross-validation function is also not parallelized, which greatly hampers its runtime.

A new, custom, k-fold cross-validation was then implemented taking these caveats into consideration; it first shuffles the input data and then proceeds to execute the folds in parallel on four concurrent processes. See listing 5 for details.

3 Results and conclusion

The results obtained were highly positive. Table 1 exposes the accuracy scores obtained for the classification tasks, *i.e.* fitting the linear multi-class support vector machines on the 48 000 training samples available for each scenario and then validating on the respective 24 000 samples available for each.

Scenario	**Accuracy Score**
Perfect position information	0.9926
Position with gaussian error - std = 0.10	0.9858
Position with gaussian error - std = 0.25	0.9556
Position with gaussian error - std = 0.40	0.9177

Table 1: Accuracy results for each scenario on the respective test datasets.

These results present
The results for the 10-fold cross-validation tasks are presented in table 2.

Scenario	**Avg. Accuracy**	**STD**
Perfect position information	0.9982	0.0003
Position with gaussian error - std = 0.10	0.9985	0.0002
Position with gaussian error - std = 0.25	0.9985	0.0002
Position with gaussian error - std = 0.40	0.9994	0.0003

Table 2: Results for the 10-fold cross-validation tasks.

Performance metrics for the fitting, validating and cross-validating of the data were also evaluated. Specifically, we recollected information on the runtime of each stage along with the CPU and RAM usage of the program, presented in table 3 and figures 1 and 2.

Task	**Avg. Time [s]**
Fit Classifier	107.78
Validate on test data	2.04
10-fold cross-validation	489.93

Table 3: Average runtime for each stage of the program.

References

[1] Grigorios Tsoumakas and Ioannis Katakis. "Multi-label classification: An overview". In: *International Journal of Data Warehousing and Mining* 3.3 (2006).

[2] Chih-Wei Hsu and Chih-Jen Lin. "A comparison of methods for multiclass support vector machines". In: *IEEE transactions on Neural Networks* 13.2 (2002), pp. 415–425.

[3] D. M. J. Tax and R. P. W. Duin. "Using two-class classifiers for multiclass classification". In: *Object recognition supported by user interaction for service robots*. Vol. 2. 2002, 124–127 vol.2. DOI: 10.1109/ICPR.2002.1048253.

[4] P. Chen and S. Liu. "An Improved DAG-SVM for Multi-class Classification". In: *2009 Fifth International Conference on Natural Computation*. Vol. 1. Aug. 2009, pp. 460–462. DOI: 10.1109/ICNC.2009.275.

[5] John C Platt, Nello Cristianini, and John Shawe-Taylor. "Large margin DAGs for multiclass classification". In: *Advances in neural information processing systems*. 2000, pp. 547–553.

[6] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[7] P. Szymański and T. Kajdanowicz. "A scikit-based Python environment for performing multi-label classification". In: *ArXiv e-prints* (Feb. 2017). arXiv: 1702.01460 [cs.LG].

[8] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed <today>]. 2001–. URL: http://www.scipy.org/.

[9] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[10] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37. eprint: http://aip.scitation.org/doi/pdf/10.1109/MCSE.2011.37. URL: http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37.

```
def cross_validation_fold(index, splits_in, splits_out):
    """
    k-fold cross-validation "fold": performs validation using
    exactly one of the splits as validation set and the rest
    of the dataset as training data.
    :param index: Index of the split to use as validation data
    :param splits_in: List of splits of the original
       dataset inputs
    :param splits_out: List of splits of the original
       dataset outputs
    :return: The accuracy score for a LinearSVC trained on
       all the splits except <index> and then validated
       on split <index>
    """
    validation_in = splits_in[index]
    validation_out = splits_out[index]
    cf = LabelPowerset(LinearSVC())

    # train on all splits except split <index>
    cf.fit(np.vstack(splits_in[:index] + splits_in[index + 1:]),
       sparse_vstack(splits_out[:index] + splits_out[index + 1:]))

    # validate on split <index>
    return validate(cf, validation_in,
                validation_out,
                return_predictions=False)

def k_fold_cross_validation(dataset, k=10):
    """
    Performs k_fold cross validation on a dataset using LinearSVCs
    :param dataset: Dataset tuple in format
    (train_in, train_out, test_in,test_out)
    :param k: Optional k-fold parameter. Default is 10.
    :return: Tuple containing the average accuracy and
       standard deviation obtained through
       cross-validation.
    """

    assert k > 1

    # label data has to be dense to be shuffled
    # in the same order as the input data
    data_in = np.vstack((dataset[0], dataset[2]))
    data_out = np.vstack((dataset[1].toarray(),
                    dataset[3].toarray()))
```

```
46
47      # shuffle data, then partition
48      data_in, data_out = shuffle_data(data_in, data_out)
49      data_in = np.split(data_in, k)
50      data_out = np.split(data_out, k)
51
52      # re-sparsify?
53      data_out = list(map(csr_matrix, data_out))
54
55      results = None
56      with Pool(processes=4) as pool:
57         results = pool.starmap(cross_validation_fold,
58                                zip(range(k),
59                                   itertools.repeat(data_in),
60                                   itertools.repeat(data_out)))
61
62      results = np.array(results)
63      return results.mean(), results.std()
```

Listing 5: Extract of the k-fold cross-validation subroutines.

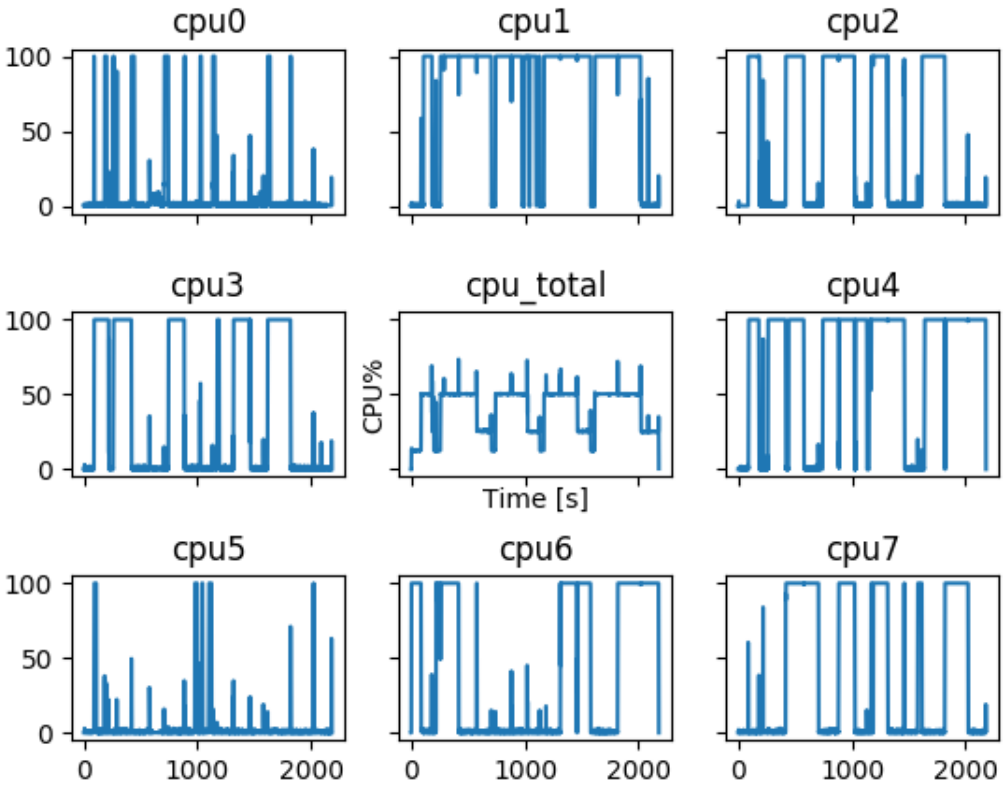

Figure 1: CPU usage for each logical core, along with the total aggregated CPU usage, over time.

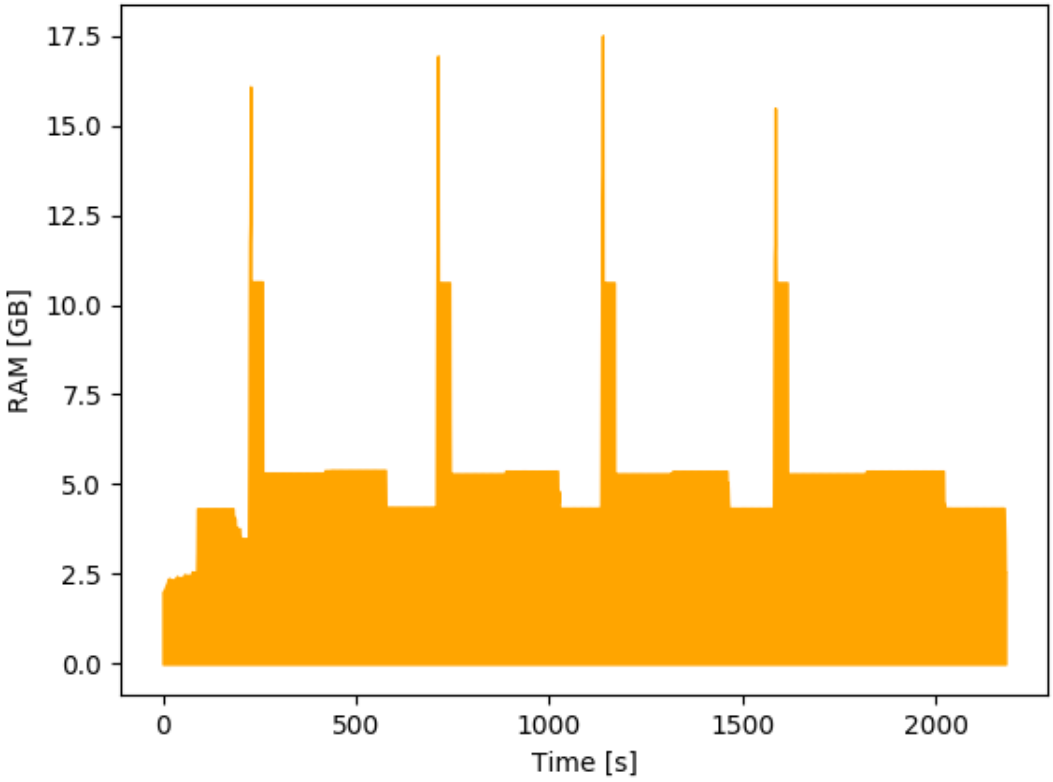

Figure 2: RAM usage during execution of the program.