Mapping Python Programs to Vectors using Recursive Neural Encodings

Benjamin Paaßen
The University of Sydney
benjamin.paassen@sydney.edu.au

Jessica McBroom
The University of Sydney
jmcb6755@sydney.edu.au

Bryn Jeffries
Grok Learning
bryn@groklearning.com

Irena Koprinska
The University of Sydney
irena.koprinska@sydney.edu.au

Kalina Yacef
The University of Sydney
kalina.yacef@sydney.edu.au

Educational data mining involves the application of data mining techniques to student activity. However, in the context of computer programming, many data mining techniques can not be applied because they require vector-shaped input, whereas computer programs have the form of syntax trees. In this paper, we present ast2vec, a neural network that maps Python syntax trees to vectors and back, thereby enabling about a hundred data mining techniques that were previously not applicable. Ast2vec has been trained on almost half a million programs of novice programmers and is designed to be applied across learning tasks *without re-training*, meaning that users can apply it without any need for deep learning. We demonstrate the generality of ast2vec in three settings. First, we provide example analyses using ast2vec on a classroom-sized dataset, involving two novel techniques, namely progress-variance projection for visualization and a dynamical systems analysis for prediction. In these examples, we also explain how ast2vec can be utilized for educational decisions. Second, we consider the ability of ast2vec to recover the original syntax tree from its vector representation on the training data and two other large-scale programming datasets. Finally, we evaluate the predictive capability of a linear dynamical system on top of ast2vec, obtaining similar results to techniques that work directly on syntax trees while being much faster (constant- instead of linear-time processing). We hope ast2vec can augment the educational data mining toolkit by making analyses of computer programs easier, richer, and more efficient.

Keywords: computer science education, computer programs, representation learning, neural networks, visualization, program vectors

1. Introduction

Techniques for analyzing and utilizing student programs have been the focus of much recent research in computer science education. Such techniques have included hint systems to provide automated feedback to students (Paaßen et al., 2018; Piech et al., 2015; Price et al., 2019; Rivers

and Koedinger, 2017), as well as visualization and search approaches to help teachers understand student behavior (McBroom et al., 2018; Nguyen et al., 2014). Considering that programming is a key skill in many fields, including science, technology, engineering, and mathematics (Denning, 2017; McCracken et al., 2001; Wiles et al., 2009), and considering the difficulty students have with learning programming (Denning, 2017; Lahtinen et al., 2005; Qian and Lehman, 2017; Robins et al., 2003), developing and expanding the range of available techniques to improve educational outcomes are of great importance.

Unfortunately, computer programs are particularly difficult to analyze for two main reasons. First, programs come in the form of raw code or syntax trees (after compilation), which few data mining techniques can handle. Instead, one first has to represent programs differently to turn them into valid input for data mining techniques (Paaßen et al., 2018). Second, the space of possible programs grows combinatorially with program length, and most programs are never written by a student, even fewer more than once (Paaßen et al., 2018; Rivers and Koedinger, 2012; Rivers and Koedinger, 2017). Accordingly, one needs to abstract from meaningless differences between programs to shrink the space and, thus, make it easier to handle with less risk of overfitting.

Several prior works have addressed both the representation and the abstraction step, often in conjunction. For example, Rivers and Koedinger (2012) have suggested semantically motivated transformations of syntax trees to remove syntactic variations that have no semantic influence (such as unreachable code or direction of binary operators). Peddycord et al. (2014) have suggested representing programs by their output rather than their syntax. Similarly, Paaßen et al. (2016) have suggested representing programs by their execution behavior. Gulwani et al. (2018) as well as Gross et al. (2014) performed a clustering of programs to achieve a representation in terms of a few discrete clusters. We point to Section 5 and to the review of McBroom et al. (2021) for a more comprehensive list. We also note that many of the possible abstraction and representation steps are not incompatible but can be combined to achieve better results (McBroom et al., 2021).

The arguably most popular representation of computer programs is pairwise tree edit distances (Zhang and Shasha, 1989). For example, Mokbel et al. (2013), Paaßen et al. (2018), Price et al. (2017), and Rivers and Koedinger (2017) have applied variations of the standard tree edit distance for processing programs. Edit distances have the advantage that they do not only quantify distance between programs but also specify which parts of the code have to be changed to transform one program into another, which can be utilized to construct hints (Paaßen et al., 2018; Price et al., 2017; Rivers and Koedinger, 2017). Additionally, many data mining techniques for visualization, clustering, regression, and classification can deal with input in terms of pairwise distances (Pekalska and Duin, 2005; Hammer and Hasenfuss, 2010). Still, a vast majority of techniques can not. For example, of 126 methods contained in the Python library scikit-learn[1], only 24 can directly deal with pairwise distances, eight further methods can deal with kernels, which require additional transformations (Gisbrecht and Schleif, 2015), and 94 only work with vector-shaped input. As such, having a vector-formed representation of computer programs enables a much wider range of analysis techniques. Further, a distance-based representation depends on a database of reference programs to compare it to. The computational complexity required for analyzing a program thus scales at least linearly with the size of the database. By contrast, a parametric model with vector-shaped input can perform operations

[1]Taken from this list https://scikit-learn.org/stable/modules/classes.html.

independent of the size of the training data. Finally, a representation in terms of pairwise distances tries to solve the representation problem for computer programs every time anew for each learning task. Conceptually, it appears more efficient to share representational knowledge across tasks. In this paper, we aim to achieve just that: to find a mapping from computer programs to vectors (and back) that generalizes across learning tasks and thus solves the representation problem ahead of time. Afterward, we only need to add a simple model specific to the analysis we wish to perform. In other words, we wish to achieve for computer programs what word embeddings like word2vec (Mikolov et al., 2013) or GloVe (Pennington et al., 2014) offer for natural language processing: a re-usable component that solves the representation problem of programs such that subsequent research can concentrate on other data mining challenges. Our technique is by no means meant to supplant all existing representations. It merely is supposed to be another tool in the toolkit for educational data mining in computer science education.

We acknowledge that we are not the first to attempt such a vector-shaped representation. Early work has utilized hand-engineered features such as program length or cyclomatic complexity (Truong et al., 2004). More recent approaches check whether certain tree patterns occur within a given syntax tree, which gives rise to a vector by counting how often each tree pattern occurs (Nguyen et al., 2014; Zhi et al., 2018; Zimmerman and Rupakheti, 2015). One can also derive a vectorial representation from pairwise edit distances using linear algebra, although the dependence on a database remains (Paaßen et al., 2018). Finally, prior works have trained neural networks to map programs to vectors (Piech et al., 2015; Alon et al., 2019). However, all these representations lose the ability to return from a program's representation to the original program. For example, while it is simple to compute the length of a program, it is impossible to identify the original program just from its length. However, such an inverse translation is crucial for the interpretability of a result. For example, it would not help the student much to tell them that they should increase the length of their program. Instead, they would rather like to know which parts of their code they have to change.

So how does one achieve a mapping from computer programs to vectors that can be inverted and still generalizes across tasks? The solution appears to lie in recent works on autoencoders for tree-structured data (Chen et al., 2018; Dai et al., 2018; Kusner et al., 2017; Paaßen et al., 2021), which train an encoder ϕ from trees to vectors and a decoder ψ from vectors back to trees such that $\phi(\psi(\hat{x}))$ is similar to $\hat{x}$ for as many trees $\hat{x}$ as possible. While training such a model requires large amounts of data and a lot of computation time (as usual in deep learning), our hope is that a trained model can then be used as a representation and abstraction device without any further need for deep learning. Instead, one can just apply vector-based data mining techniques – e.g., any of the methods in scikit-learn – to the vector representations of computer programs and transform any vector back into a program if desired.

In this paper, we present ast2vec, an autoencoder trained on almost half a million beginner Python programs that encodes beginner Python programs as 256-dimensional vectors *and* decodes vectors back to syntax trees. Both encoding and decoding are guided by the grammar of the Python programming language, thereby ensuring that decoded trees are syntactically correct. The model and its source code are completely open-source and are available at `https://gitlab.com/bpaassen/ast2vec/`.

The overarching question guiding our research is: Does ast2vec generalize across learning tasks and across different data mining techniques? To this end, we test ast2vec in three settings. First, we utilize ast2vec for a variety of analyses on a classroom-sized dataset and interpret the results in light of educational decisions. This test shows the utility and generality qualita-

tively. Second, we quantitatively investigate the autoencoding error of ast2vec on its training data and on two similarly large datasets of novice programs from subsequent years. In all cases, we observe that small trees usually get autoencoded correctly whereas errors may occur for larger trees. Finally, we investigate the performance of a simple linear regression model on top of ast2vec for predicting the next step of students in a learning task and observe that we achieve comparable results to established techniques while being both more space- and more time-efficient.

This paper is set out as follows: Section 2 begins by utilizing ast2vec for a variety of data mining analyses on a classroom-sized dataset to illustrate its utility for educational applications. Section 3 then specifies the details of ast2vec, as well as the details of two analysis techniques used in Section 2, namely progress-variance projections and dynamic analyses. Next, Section 4 provides an evaluation of ast2vec on large-scale datasets, including an investigation of its autoencoding performance and predictive capabilities. Finally, Section 5 describes related work in more detail, and Section 6 concludes with a summary of the main ideas and findings of the paper and a discussion of future directions.

2. Example analyses using ast2vec

Before introducing the details of ast2vec, we perform example analyses to illustrate that the vector representation of ast2vec can be useful for many educational applications. In particular, we perform four example analyses using ast2vec on a classroom-sized dataset: 1) visualizing student work and progress on a task, 2) modeling student behavior, 3) clustering student programs, and 4) outlier detection.

Note that we have used a synthetic dataset for these examples because we wanted to make all of the code and data used in this section publicly available. In Section 4, we cover experiments on large-scale, real datasets.

2.1. Dataset Construction

To construct the dataset used in these example analyses, we manually simulated ten students attempting to solve the following introductory programming task:

> Write a program that asks the user, "What are your favourite animals?". If their response is "Quokkas", the program should print "Quokkas are the best!". If their response is anything else, the program should print "I agree, x are great animals.", where x is their response.

The reference solution for the task is:

```
x = input("What are your favourite animals? ")
if x == "Quokkas":
    print("Quokkas are the best!")
else:
    print(f"I agree, {x} are great animals.")
```

We simulated the development process by starting with an empty program, writing a partial program, receiving feedback from automatic test cases for this partial program, and revising the program in response to the test feedback until a correct solution was reached. Our four simulated

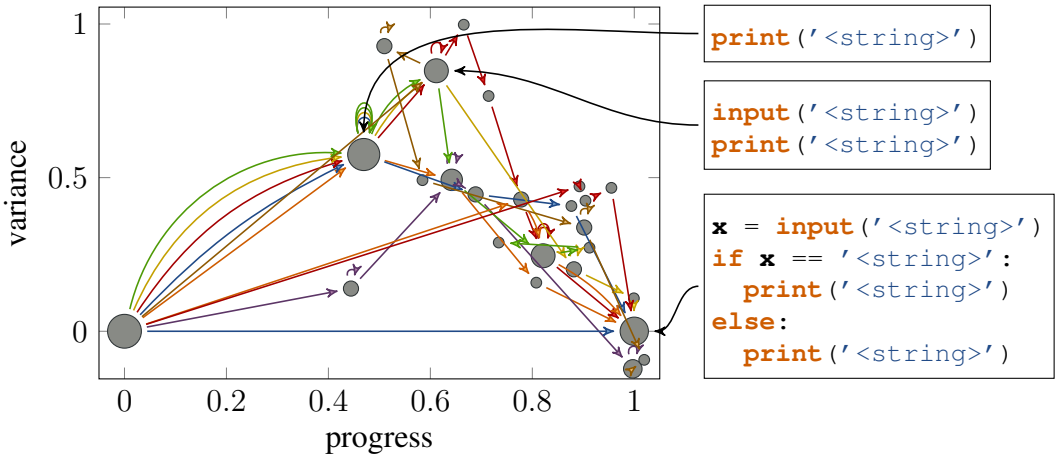

Figure 1: A progress-variance plot of the example task. The special points $(0, 0)$ and $(1, 0)$ correspond to the empty program and the reference solution, respectively. The size of points corresponds to their frequency in the data. Arrows indicate student motion. Different students are plotted in different colors.

unit tests checked 1) whether the first line of output was "What are your favourite animals?", and 2-4) whether the remaining output was correct for the inputs "Quokkas", "Koalas" and "Echidnas", respectively.

Overall, the dataset contains 58 (partial) programs, and these form $N = 25$ unique syntax trees after compilation. Since ast2vec converts program trees to $n = 256$ dimensional vectors, this means the data matrix $\boldsymbol{X} \in \mathbb{R}^{N \times n}$ after encoding all programs is of size $N \times n = 25 \times 256$.

2.2. Application 1: Visualizing Student Work and Progress

As a first example application of ast2vec, we consider a visualization of student progress from the empty program to the correct solution. The purpose of such a visualization is to give educators a concise overview of typical paths towards the goal, where students tend to struggle, as well as how many typical strategies exist (McBroom et al., 2021).

In order to create such a visualization, we combine ast2vec with a second technique that we call *progress-variance projection*. Our key idea is to construct a linear projection from 256 dimensions to two dimensions, where the first axis captures the progress from the empty program towards the goal and the second axis captures as much of the remaining variance between programs as possible. A detailed description of this approach is given in Section 3.2.

Figure 1 shows the result of applying this process to the sample data. Each circle in the plot represents a unique syntax tree in the data, with larger circles representing more frequent trees and the three most common trees shown on the right. Note that $(0, 0)$ represents the empty program and $(1, 0)$ represents the solution. In addition, the arrows indicate student movement between programs, with different colors representing different students. More specifically, an arrow is drawn from vector $\vec{x}$ to $\vec{y}$ if the student made a program submission corresponding to vector $\vec{x}$ followed by a submission corresponding to vector $\vec{y}$.

Without the axis labels, this plot is similar to the interaction networks suggested by Barnes et al. (2016), which have already been shown to provide useful insights into student learning

(McBroom et al., 2018). However, in contrast to interaction networks, our progress-variance projection additionally maps vectors to meaningful locations in space. In particular, x-axis corresponds to the (linear) progress toward the solution, whereas the y-axis corresponds to variation between student programs that is orthogonal to the progress direction. This visualization can provide an intuitive overview of the types of programs students submit and how they progress through the exercise.

In more detail, we can interpret the visualization as follows. While there is a lot of difference on the fine-grained level, there appears to be a consistent path towards the solution, which follows an arc from the origin upwards and then down towards the correct solution. Interestingly, though, we also see several students who take a slight detour towards location $(0.65, 0.8)$ in Figure 1 before moving back towards the correct solution. This detour corresponds to programs which ask the user for input and then print something but which do not store the user's input in a variable. This may be an indication of an unhelpful direction that should be addressed with teacher feedback. Further evidence in this regard is the fact that a student moved away from the correct solution, whereas all other arrows in the plot are forward motions in the progress axis. The visualization also suggests that location $(0.5, 0.55)$ (corresponding to programs of the form `print('string')`) is a decision point where students either get stuck (indicated by the loops in the plot), move to the detour, or move towards the correct solution. A hint at this location may put students on the right track towards a correct solution.

In summary, we saw how ast2vec can be used to visualize student progress through a programming task, how we can interpret such a visualization, and how we can utilize our interpretation for educational interventions. Next, we consider student motion through the space of possible programs in more detail.

2.3. Application 2: Dynamical analysis

In Figure 1, we observed that there appears to be a consistent path towards the correct solution, namely an arc beginning at the empty program at $(0, 0)$, then rising in the y-axis, before approaching the reference solution at $(1, 0)$. This raises the question: Can we capture this general motion of students in a simple, dynamical system? Such a dynamical system could be useful for hint generation. If a student does not know how to continue, we can provide a prediction of how students would generally proceed, namely following the arc in the plot. Then, we can use ast2vec to decode the predicted position back into a syntax tree and use an edit distance to construct the hint, similar to prior techniques (Paaßen et al., 2018; Price et al., 2017; Rivers and Koedinger, 2017).

In Section 3.3, we describe a technique to learn a linear dynamical system from data. Figure 2 illustrates this system for our case. In particular, the orange arrows indicate how a point at the origin of the arrow would be moved by the dynamical system. As we can see, the dynamical system does indeed capture the qualitative behavior we observe in Figure 1, namely the arc-shaped motion from the empty program towards the reference solution. We can verify this finding by simulating a new trace that follows our dynamical system. In particular, we start at the empty program $\vec{x}_0$ and then follow the direction of the orange arrows until the location does not change much anymore[2]. The resulting motion is plotted in blue. We further decode all steps $\vec{x}_t$ using ast2vec and inspect the resulting syntax tree. These trees are shown in Figure 2 on the right. We observe that the simulated trace corresponds to reasonable student motion, namely to

[2]More precisely, we apply Euler steps in the dynamical system with step size 1.

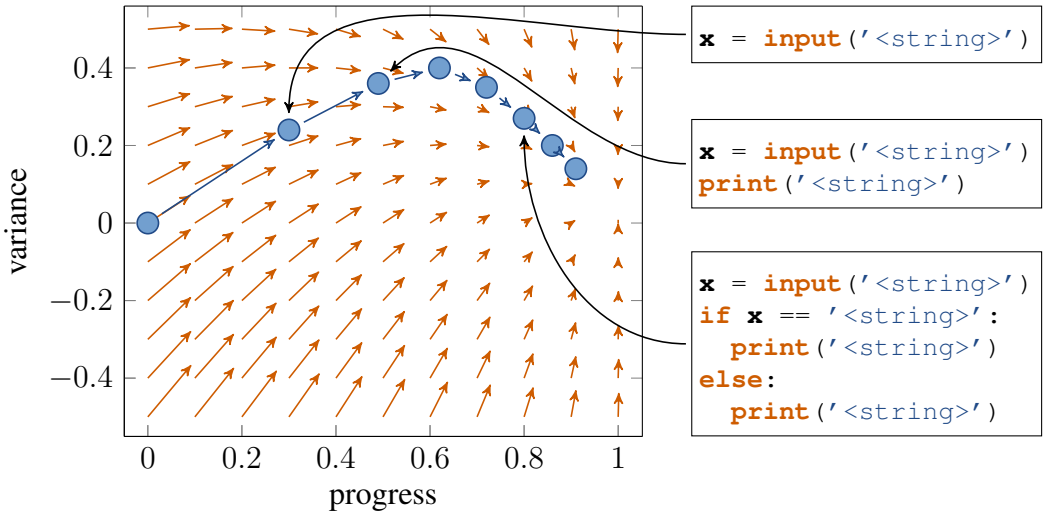

Figure 2: A linear dynamical system f, which has a single attractor at the correct solution and approximates the motion of students through the space of programs (orange). Arrows are scaled with factor 0.3 for better visibility. An example trace starting at the empty program and following the dynamical system is shown in blue. Whenever the decoded program changes along the trace, we show the code on the right. The coordinate system is the same as in Figure 1.

first ask the user for input, then add a single print statement, and finally to extend the solution with an if-statement that can distinguish between the input 'Quokka' and all other inputs.

As mentioned above, we can use this dynamical system for hint generation. For example, if a student is stuck at a program of the shape `print('string')`, we could ask the dynamical system to suggest a next step that does not reveal the correct solution but gets closer to it, which would also avoid detours. Additionally, we can use the dynamical system to identify atypical student motion. For example, we can overlay Figure 1 on top of Figure 2 and compare the orange arrows with actual student motion. In our case, this method reveals that the cluster at location $(0.65, 0.8)$ in Figure 1 is, indeed, atypical and can be interpreted as a detour, which is consistent with our prior interpretation. Finally, the dynamical system itself may reveal common errors. If there are locations where arrows point away from the correct solution or where the system is changing rapidly, this indicates that a large part of the student population does not know how to continue in this situation. In that case, educators may wish to inspect the corresponding program shape and design feedback that helps students get back on track.

2.4. Application 3: Clustering

When inspecting Figure 1, we also notice that student programs seem to fall into clusters, namely one cluster around the correct solution, one cluster to the top left of it, and further minor clusters. Clustering student data has been shown to be useful in a wide range of educational data mining applications (Baker and Yacef, 2009), such as revealing differences between subgroups of the student population (McBroom et al., 2020), or making it easier to design feedback by applying the same feedback strategy for every program in the same cluster (Gross et al., 2014; Glassman et al., 2015).

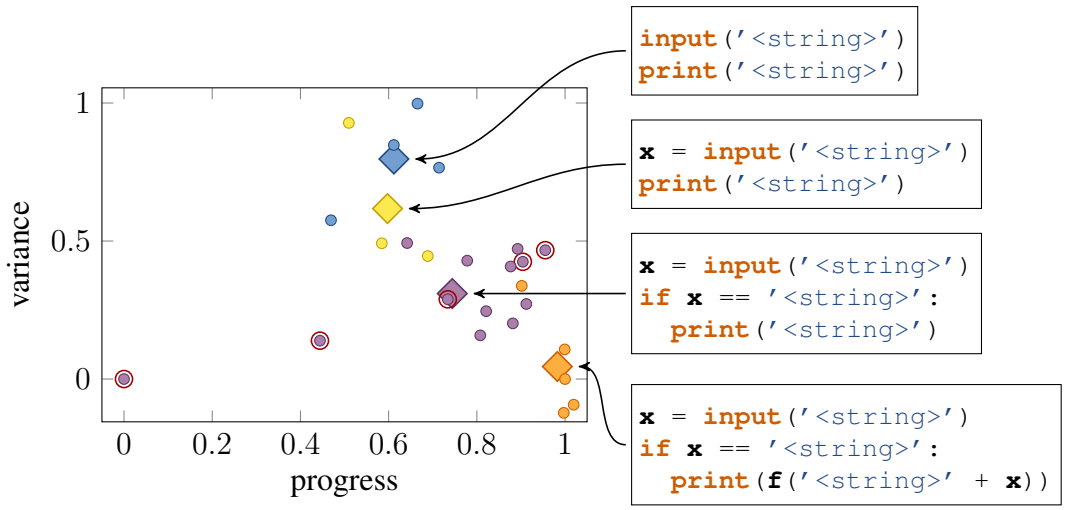

Figure 3: A visualization of Gaussian mixture clusters of programs with color indicating cluster assignment. Cluster means are shown as diamonds. Outliers are highlighted with a red ring. The coordinate system is the same as in Figure 1.

Thanks to the vectorial representation provided by ast2vec, we can apply a wide range of clustering techniques. For this example, we use a Gaussian mixture model, which also provides us with probability (Dempster et al., 1977). In particular, a Gaussian mixture model expresses the data as a mixture of K Gaussians

$$p(\vec{x}) = \sum_{k=1}^{K} p(\vec{x}|k) \cdot p(k),$$

where $p(\vec{x}|k)$ is a Gaussian density and $p(k)$ is the prior for the kth Gaussian. The mean, covariance, and prior of each Gaussian are fitted to the data. After training, we assign data vectors $\vec{x}_i$ to clusters by evaluating the posterior probability $p(k|\vec{x}_i)$ and assigning the Gaussian k with the highest probability. Note that, prior to clustering, we perform a principal component analysis (PCA) because Gaussian densities are based on distances, which degenerate in high dimensions (Aggarwal et al., 2001). We set the PCA to preserve 95% of the data variance, which yielded 12 dimensions. Then, we cluster the data via a Gaussian mixture model with $K = 4$ Gaussians.

Figure 3 shows the resulting clustering with clusters indicated by color and cluster means by diamonds. We decode the cluster means back into syntax trees using ast2vec. This enables us to interpret the clusters. The yellow, purple, and orange cluster capture the 'typical' path towards a correct solution. Students first ask a user for input, store the input in a variable, and print a reaction in return (yellow cluster). Then, they add an 'if' statement (purple cluster), and finally, they integrate the user's input into the output, yielding the structure of a correct solution (orange cluster). The blue cluster corresponds to the detour we found before. If students enter this cluster, we may want to provide a hint based on our dynamical system above, provide them with a tailored message prepared by a teacher, or recommend another learning task that explains the 'input' statement in more detail.

We could also use this clustering to analyze differences between student populations in more

detail, be it for the purpose of personalization or to make the learning environment more equitable across groups (McBroom et al., 2020). For example, if we find that the blue cluster is more common in younger students, this could indicate that younger students may benefit from additional support to avoid this detour.

2.5. Application 4: Outlier Detection

An additional benefit of a probability-based clustering is that we can use it directly to detect "outliers", meaning programs that are atypical and can not be captured well by a clustering. Detecting such outliers can be useful to find cases where our current educational strategies fail and where we need new interventions.

Figure 3 highlights all points with a red circle that received particularly low probability $p(\vec{x})$ by our Gaussian mixture model[3]. Unsurprisingly, one outlier is the empty program, which is distant from every non-empty program. The remaining outliers are more instructive. In particular, the outlier in the center of the plot corresponds to an input statement without a print statement, which is an unusual path towards the solution. In this dataset, it is more common to write the print statement first. Accordingly, we may want to check that we do not accidentally provide feedback to these students that presumes they should have a print statement already.

The two outliers around $(0.9, 0.45)$ correspond to the following program shape:

```
print('What are your favourite animals? ')
animals = input()
if animals == 'Quokkas':
    print('Quokkas are the best!')
else:
    print(f'I agree, {animals} are great animals.')
```

Here, the question for the favorite animals is posed as a print statement, and the input is requested with a separate command, which is a misunderstanding of how `input` works. Here, we may wish to intervene with feedback that explains `input` to the student.

The final outlier is the program:

```
animals = input('What are your favourite animals? ')
if animals == 'Koalas':
    print('I agree, Koalas are great animals.')
elif animals == 'Echidnas':
      print('I agree, Echidnas are great animals.')
else:
    print('Quokkas are the best!')
```

This program does pass all our test cases but does not adhere to the "spirit" of the task, because it does not generalize to new inputs. Such an outlier may instruct us that we need to change our test cases to be more general, e.g., by using a hidden test with a case that is not known to the student.

[3] We defined 'particularly low' as having a log probability of at most half the average when normalizing by the least likely sample.

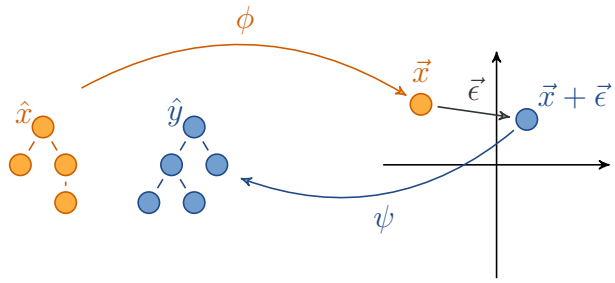

Figure 4: A high-level illustration of the (denoising) autoencoder framework. Syntax tree $\hat{x}$ (left) gets encoded as a vector $\vec{x} = \phi(\hat{x})$ (right). We then add noise $\vec{\epsilon}$ and decode back to a tree $\hat{y} = \psi(\vec{x} + \vec{\epsilon})$. If $\hat{x} \neq \hat{y}$, we adjust the parameters of both encoder and decoder to increase the probability that $\vec{x} + \vec{\epsilon}$ is decoded into $\hat{x}$ instead of $\hat{y}$.

2.6. Further analyses

This concludes our example analyses using ast2vec. We note that further types of data analysis are possible as well. For example, we could train a classifier that receives the vector representation of a program as input and tries to predict student performance (Koprinska et al., 2015) or whether the student could benefit from a certain feedback message for this task (Piech et al., 2015). One could also use a clustering as a basis for an interaction network, enabling further hint strategies (Barnes et al., 2016). In general, we hope that ast2vec proves itself useful as a tool in a broad range of educational data mining pipelines for computer science education.

In the remainder of this paper, we will explain how ast2vec works in more detail and evaluate it on real large-scale data.

3. Methods

In this section, we describe the methods employed in this paper in more detail. In Section 3.1, we provide a summary of the autoencoder approach we used for ast2vec. In Section 3.2, we describe the progress-variance projection that we used to generate 2D visualizations of student data. Finally, in Section 3.3, we explain how to learn linear dynamical systems from student data.

3.1. The Architecture of ast2vec

Our neural network ast2vec is an instance of a recursive tree grammar autoencoder model as proposed by Paaßen et al. (2021). In this section, we describe this approach as we used it for ast2vec. If readers are interested in the general approach (and more technical details), we recommend the original paper.

On a high level, ast2vec is a so-called *autoencoder*, i.e., a combination of an encoder $\phi : \mathcal{X} \to \mathbb{R}^n$ from trees to vectors and a decoder $\psi : \mathbb{R}^n \to \mathcal{X}$ from vectors to trees, such that $\psi(\phi(\hat{x}))$ is equal to $\hat{x}$ for as many trees $\hat{x}$ as possible.

More precisely, we apply the variational autoencoding (VAE) framework of Kingma and Welling (2013). In this framework, we train the neural net to not only work on the training data but also if we add a small amount of noise to the vector code before decoding it again (refer

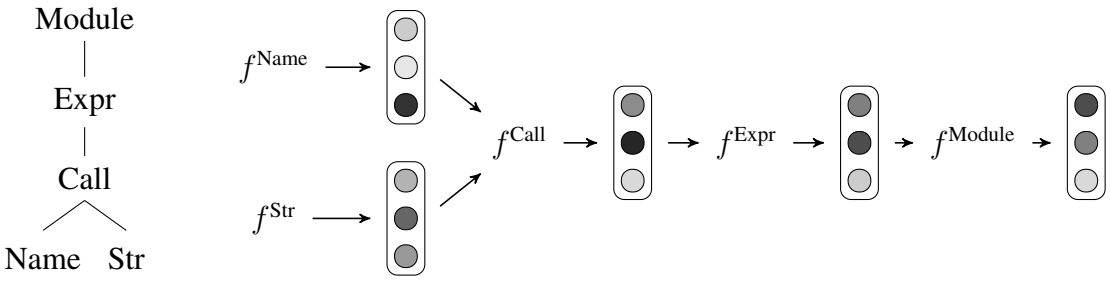

Figure 5: An illustration of the encoding process into a vector with $n = 3$ dimensions. We first use the Python compiler to transform the program `print('Hello, world!')` into the abstract syntax tree Module(Expr(Call(Name, Str))) (left). Then, from left to right, we apply the recursive encoding scheme, where arrows represent the direction of computation and traffic light-elements represent three-dimensional vectors, where color indicates the value of the vector in that dimension.

also to Figure 4). Further, we force the distribution of vector codes to be a standard normal distribution.

In the next paragraphs, we describe the encoder ϕ and the decoder ψ of ast2vec.

ENCODING. The first step of our encoding is to use the Python compiler to generate an abstract syntax tree for the program. Let $x(\hat{y}_1, \ldots, \hat{y}_K)$ be such an abstract syntax tree, where x is the root syntactic element and $\hat{y}_1, \ldots, \hat{y}_K$ are its K child subtrees. For an example of such a syntax tree, refer to Figure 5 (left). Our encoder ϕ is then defined as follows.

$$\phi\Big(x(\hat{y}_1, \ldots, \hat{y}_K)\Big) := f^x\Big(\phi(\hat{y}_1), \ldots, \phi(\hat{y}_K)\Big), \tag{1}$$

where $f^x : \mathbb{R}^{K \times n} \to \mathbb{R}^n$ is a function that takes the vectors $\phi(\hat{y}_1), \ldots, \phi(\hat{y}_K)$ for all children as input and returns a vector for the entire tree, including the syntactic element x and all its children. Because Equation 1 is recursively defined, we also call our encoding *recursive*. Figure 5 shows an example encoding for the program `print('Hello, world!')` with $n = 3$ dimensions. We first use the Python compiler to translate this program into the abstract syntax tree Module(Expr(Call(Name, Str))) and then apply our recursive encoding scheme. In particular, recursively applying Equation 1 to this tree yields the expression $f^{\text{Module}}(f^{\text{Expr}}(f^{\text{Call}}(f^{\text{Name}}(), f^{\text{Str}}())))$. We can evaluate this expression by first computing the leaf terms $f^{\text{Name}}()$ and $f^{\text{Str}}()$, which do not require any inputs because they have no children. This yields two vectors, one representing Name and one representing Str, respectively. Next, we feed these two vectors into the encoding function f^{Call}, which transforms them into a vector code of the subtree Call(Name, Str). We feed this vector code into f^{Expr}, whose output we feed into f^{Module}, which in turn yields the overall vector encoding for the tree. Note that our computation follows the structure of the tree bottom-up, where each encoding function receives exactly as many inputs as it has children.

A challenge in this scheme is that we have to know the number of children K of each syntactic element x in advance to construct a function f^x. Fortunately, the grammar of the programming language[4] tells us how many children are permitted for each syntactic element.

[4]The grammar for Python can be found at: https://docs.python.org/3/library/ast.html.

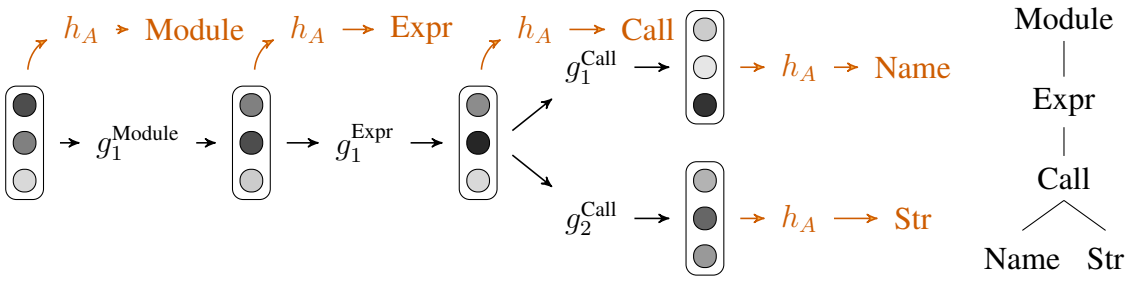

Figure 6: An illustration of the decoding process of a vector with $n = 3$ dimensions into an abstract syntax tree. We first plug the current vector into a scoring function h, based on which we select the current syntactic element x (shown in orange). After this step, we feed the current vector into decoding functions $g_1^x, \ldots, g_K^x$, one for each child a syntactic element x expects. This yields the next vectors, for which the same process is applied again until all remaining vectors decode into leaf elements.

For example, an `if` element in the Python language has three children, one for the condition, one for the code that gets executed if the condition evaluates to `True` (the 'then' branch), and one for the code that gets executed if the condition evaluates to `False` (the 'else' branch).

Leaves, like Str or Name, are an important special case. Since these have no children, their encoding is a constant vector $f^x \in \mathbb{R}^n$. This also means that we encode all possible strings as the same vector (the same holds for all different variable names or all different numbers). Incorporating content information of variables in the encoding is a topic for future research.

Another important special case are lists. For example, a Python program is defined as a list of statements of arbitrary length. In our scheme, we encode a list by adding up all vectors inside it. The empty list is represented as a zero vector.

Note that, up to this point, our procedure is entirely general and has nothing to do with neural nets. Our approach becomes a (recursive) neural network because we implement the encoding functions f^x as neural networks. In particular, we use a simple single-layer feedforward network for the encoding function f^x of the syntactic element x:

$$f^x(\vec{y}_1, \ldots, \vec{y}_K) = \tanh\left(\boldsymbol{U}_1^x \cdot \vec{y}_1 + \ldots + \boldsymbol{U}_K^x \cdot \vec{y}_K + \vec{b}^x\right), \tag{2}$$

where $\boldsymbol{U}_k^x \in \mathbb{R}^{n \times n}$ is the matrix that decides what information flows from the kth child to its parent and $\vec{b}^x \in \mathbb{R}^n$ represents the syntactic element x. These $\boldsymbol{U}$ matrices and the $\vec{b}$ vectors are the parameters of the encoder that need to be learned during training.

Importantly, this architecture is still relatively simple. If one would aim to optimize autoencoding performance, one could imagine implementing f^x instead with more advanced neural networks, such as a Tree-LSTMs (Chen et al., 2018; Dai et al., 2018; Tai et al., 2015). This is a topic for future research.

DECODING. To decode a vector $\vec{x}$ recursively back into a syntax tree $x(\hat{y}_1, \ldots, \hat{y}_K)$, we have to make $K + 1$ decisions. First, we have to decide the syntactic element x. Then, we have to decide the vector codes $\vec{y}_1, \ldots, \vec{y}_K$ for each child. For the first decision we set up a function h_A which computes a numeric score $h_A(x|\vec{x})$ for each possible syntactic element x from the vector

$\vec{x}$. We then select the syntactic element x with the highest score. Importantly, the A in the index of h refers to the fact that we guide our syntactic decision by the Python grammar. In particular, we have a separate function h_A for every grammar symbol that has one output for every syntactic element permitted by the symbol A. For simplicity, we do not discuss the details of grammar rules here but point the interested reader to Paaßen et al. (2021).

Once we know the syntactic element, the Python grammar tells us the number of children K, i.e., how many child codes we need to generate. Accordingly, we use K decoding functions $g_k^x : \mathbb{R}^n \to \mathbb{R}^n$ which tell us the vector code for each child based on the parent code $\vec{x}$. The precise definition of the decoding procedure is:

$$\psi(\vec{x}) = x\Big(\psi(\vec{y}_1), \ldots, \psi(\vec{y}_k)\Big), \tag{3}$$

where

$$x = \arg\max_y h_A(y|\vec{x}) \qquad \text{and} \qquad \vec{y}_k = g_k^x(\vec{x}) \qquad \text{for all } k \in \{1, \ldots, K\}$$

In other words, we first use h_A to choose the current syntactic element x with maximum score $h_A(x|\vec{x})$, use the decoding functions $g_1^x, \ldots, g_K^x$ to compute the neural codes for all children, and proceed recursively to decode child subtrees.

An example of the decoding process is shown in Figure 6. As input, we receive some vector $\vec{x}$, which we first feed into the scoring function h_A. As the Python grammar requires that each program begins with a Module, the only score higher than $-\infty$ is $h(\text{Module}|\vec{x})$, meaning that the root of our generated tree is Module. Next, we feed the vector $\vec{x}$ into the decoding function g_1^{Module}, yielding a vector $g_1^{\text{Module}}(\vec{x})$ to be decoded into the child subtree. With this vector, we re-start the process, i.e. we feed the vector $g_1^{\text{Module}}(\vec{x})$ into our scoring function h_A, which this time selects the syntactic element Expr. We then generate the vector representing the child of Expr as $\vec{y} = g_1^{\text{Expr}}(g_1^{\text{Module}}(\vec{x}))$. For this vector, h_A selects Call, which expects two children. We obtain the vectors for these two children as $g_1^{\text{Call}}(\vec{y})$ and $g_2^{\text{Call}}(\vec{y})$. For these vectors, h_A selects Name and Str, respectively. Neither of these has children, such that the process stops, leaving us with the tree Module(Expr(Call(Name, Str))).

Again, we note the special case of lists. To decode a list, we use a similar scheme, where we let a function h_{A^*} make a binary choice whether to continue the list or not, and we use a decoding function g_k^x to decide the code for the next list element.

We implement the decoding functions g_k^x with single-layer feedforward neural networks as in Equation 2. A special case are the decoding functions for list-shaped children, which we implement as recurrent neural networks, namely gated recurrent units (Cho et al., 2014). Again, we note that one could choose to implement all decoding functions g_k^x as recurrent networks, akin to the decoding scheme suggested by Chen et al. (2018). Here, we opt for a simple version and leave the extension to future work.

For the scoring functions h_A, we use a linear layer with n inputs and as many outputs as there are permitted syntactic elements for grammar symbol A.

We note that ast2vec only decodes to syntax trees and does not include variable names, numbers, or strings. However, it is possible to train a simple support vector machine classifier which maps the vector code for each node of the tree during decoding to the variable or function it should represent. This is how we obtain the function names and variables in our Figures like in Figure 1. We provide the source code for training such a classifier as part of our code distribution at `https://gitlab.com/bpaassen/ast2vec/`.

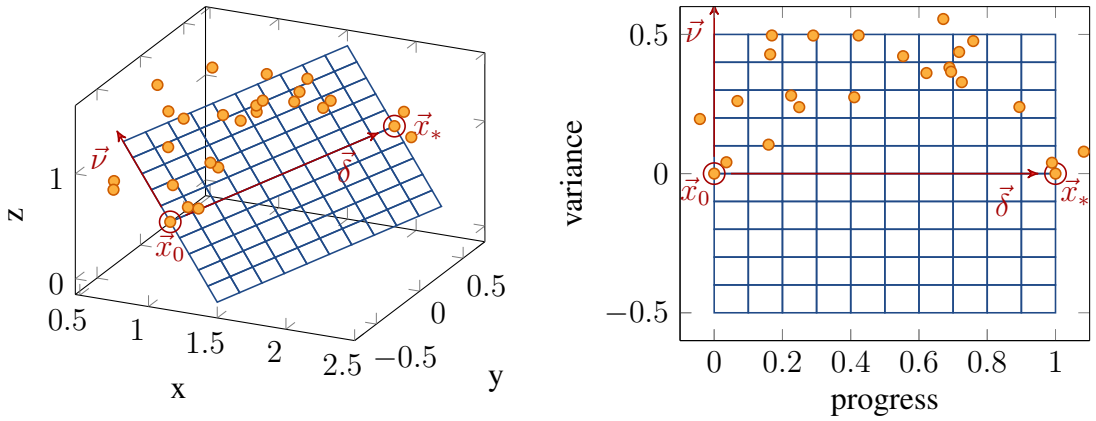

Figure 7: An illustration of the progress-variance plotting technique. Left: The high-dimensional space (here 3D) where orange points indicate program encodings. The encoding for the empty program and the reference solution are highlighted in red and annotated with $\vec{x}_0$ and $\vec{x}_*$, respectively. The x-axis of our progress-variance plot is the vector $\vec{\delta} = \vec{x}_* - \vec{x}_0$. The y-axis is the orthogonal axis $\vec{\nu}$, which covers as much variance as possible of the point distribution. Right: The 2D-dimensional coordinate system with all points projected down to 2D.

TRAINING. For the details of the training procedure on a large-scale Python dataset, we refer the reader to Appendix A. Importantly, we do not expect teachers to perform this training procedure themselves. Instead, we propose to use the pre-trained ast2vec model as a general-purpose component without retraining (refer to Section 2). In the following sections, we explain two example techniques that are possible thanks to the vectorial representation achieved by ast2vec.

3.2. PROGRESS-VARIANCE PROJECTIONS

In this section, we present a novel technique for visualizing student programs, the *progress-variance projection*. This technique, which requires a vector encoding as provided by ast2vec, involves mapping program vectors to a two-dimensional representation where the first axis captures the progress between an empty program and the solution and where the second axis captures the variation orthogonal to the progress axis. In particular, we map our vectors onto the plane spanned by the following two orthogonal vectors:

1. $\vec{\delta} = \vec{x}_* - \vec{x}_0$, where $\vec{x}_*$ and $\vec{x}_0$ are the encodings of the solution and empty program respectively. This is used as the x-axis of the plot and captures progress towards or away from the solution.

2. $\vec{\nu}$, which is chosen as the unit vector orthogonal to $\vec{\delta}$ that preserves as much variance in the dataset as possible. Note that this setup is similar to principal component analysis (Pearson, 1901), but with the first component fixed to $\vec{\delta}$.

The full algorithm to obtain the 2D representation of all vectors is shown in Algorithm 1.

Algorithm 1 An algorithm to map a data matrix of program encodings $\boldsymbol{X} \in \mathbb{R}^{N \times n}$ to a 2D progress-variance representation $\boldsymbol{Y} \in \mathbb{R}^{N \times 2}$ based on an encoding for the empty program $\vec{x}_0 \in \mathbb{R}^n$ and an encoding for the reference solution $\vec{x}_* \in \mathbb{R}^n$.

$\vec{\delta} \leftarrow \vec{x}_* - \vec{x}_0$.
$\hat{\delta} \leftarrow \vec{\delta} / \|\vec{\delta}\|$.
$\hat{\boldsymbol{X}} \leftarrow \boldsymbol{X} - \vec{x}_0$. ▷ Row-wise subtraction
$\hat{\boldsymbol{X}} \leftarrow \hat{\boldsymbol{X}} - \hat{\boldsymbol{X}} \cdot \hat{\delta} \cdot \hat{\delta}^T$. ▷ Project $\vec{\delta}$ out, i.e. $\hat{\boldsymbol{X}}$ lives in the orthogonal space to δ
$\boldsymbol{C} \leftarrow \hat{\boldsymbol{X}}^T \cdot \hat{\boldsymbol{X}}$. ▷ Quasi Covariance matrix
$\vec{\nu} \leftarrow$ eigenvector with largest eigenvalue of $\boldsymbol{C}$.
$\boldsymbol{Y} \leftarrow (\boldsymbol{X} - \vec{x}_0) / \|\vec{\delta}\| \cdot (\hat{\delta}, \vec{\nu})$. ▷ Map to 2D
return $\boldsymbol{Y}$.

Once $\vec{\delta}$ and $\vec{\nu}$ are known, we can translate any high-dimensional vector $\vec{x} \in \mathbb{R}^n$ to a 2D version $\vec{y}$ and back via the following equations:

$$\vec{y} = \begin{pmatrix} \hat{\delta} & \vec{\nu} \end{pmatrix}^T \cdot (\vec{x} - \vec{x}_0) \;/\; \|\vec{\delta}\| \qquad \text{(high to low projection)} \tag{4}$$

$$\vec{x}' = \begin{pmatrix} \hat{\delta} & \vec{\nu} \end{pmatrix} \cdot \vec{y} \;\times\; \|\vec{\delta}\| + \vec{x}_0, \qquad \text{(low to high embedding)} \tag{5}$$

where $\hat{\delta} = \vec{\delta} / \|\vec{\delta}\|$ is the unit vector parallel to $\vec{\delta}$ and $\begin{pmatrix} \hat{\delta} & \vec{\nu} \end{pmatrix}$ is the matrix with $\hat{\delta}$ and $\vec{\nu}$ as columns.

In general, $\vec{x}'$ will not be equal to the original $\vec{x}$ because the 2D space can not preserve all information in the $n = 256$ dimensions. However, our special construction ensures that the empty program $\vec{x}_0$ corresponds exactly to the origin $(0, 0)$ and that the reference solution $\vec{x}_*$ corresponds exactly to the point $(1, 0)$, which can be checked by substituting $\vec{x}_0$ and $\vec{x}_*$ into the equations above. In other words, the x-axis represents linear progress in the coding space toward the goal, and the y-axis represents variance orthogonal to progress. An example of the geometric construction of our progress-variance plot is shown in Figure 7. In this example, we project a three-dimensional dataset down to 2D.

3.3. Learning Dynamical Systems

In this section, we describe how to learn a linear dynamical system that captures the motion of students in example data. We note that we are not the first to learn (linear) dynamical systems from data (Campi and Kumar, 1998; Hazan et al., 2017). However, to our knowledge, we are the first to apply dynamical systems learning for educational data mining purposes.

In particular, we consider a linear dynamical system f, which predicts the next step of a student at location $\vec{x}$:

$$f(\vec{x}) = \vec{x} + \boldsymbol{W} \cdot (\vec{x}_* - \vec{x}), \tag{6}$$

where $\vec{x}_*$ is a correct solution to the task and $\boldsymbol{W}$ is a matrix of parameters to be learned.

This form of the dynamical system is carefully chosen to ensure two desirable properties. First, the reference solution is a guaranteed fixed point of our system, i.e. we obtain $f(\vec{x}_*) = \vec{x}_*$. Indeed, we can prove that this system has the reference solution $\vec{x}_*$ as unique stable attractor if $\boldsymbol{W}$ has sufficiently small eigenvalues (refer to Theorem 1 in Appendix B for details). This is desirable because it guarantees that the default behavior of our system is to move towards the correct solution.

Second, we can *learn* a matrix $\boldsymbol{W}$ that best describes student motion via simple linear regression. In particular, let $\vec{x}_1, \ldots, \vec{x}_T$ be a sequence of programs submitted by a student in their vector encoding provided by ast2vec. Then, we wish to find the matrix $\boldsymbol{W}$ which best captures the dynamics in the sense that $f(\vec{x}_t)$ should be as close as possible to $\vec{x}_{t+1}$ for all t. More formally, we obtain the following minimization problem:

$$\min_{\boldsymbol{W}} \quad \sum_{t=1}^{T-1} \|f(\vec{x}_t) - \vec{x}_{t+1}\|^2 + \lambda \cdot \|\boldsymbol{W}\|_{\mathcal{F}}^2, \tag{7}$$

where $\|\boldsymbol{W}\|_{\mathcal{F}}$ is the Frobenius norm of the matrix $\boldsymbol{W}$ and where $\lambda > 0$ is a parameter that can be increased to ensure that the reference solution remains an attractor. This problem has the closed-form solution

$$\boldsymbol{W} = \left(\boldsymbol{X}_{t+1} - \boldsymbol{X}_t\right)^T \cdot \boldsymbol{X}_t \cdot \left(\boldsymbol{X}_t^T \cdot \boldsymbol{X}_t + \lambda \cdot \boldsymbol{I}\right)^{-1}, \tag{8}$$

where $\boldsymbol{X}_t = (\vec{x}_* - \vec{x}_1, \ldots, \vec{x}_* - \vec{x}_{T-1})^T \in \mathbb{R}^{T-1 \times n}$ is the concatenation of all vectors in the trace up to the last one and $\boldsymbol{X}_{t+1} = (\vec{x}_* - \vec{x}_2, \ldots, \vec{x}_* - \vec{x}_T)^T \in \mathbb{R}^{T-1 \times n}$ is the concatenation of all successors. Refer to Theorem 2 in Appendix B for a proof.

As a side note, we wish to highlight that this technique can readily be extended to multiple reference solutions by replacing $\vec{x}_*$ in Equations 6 and 8 with the respective closest correct solution to the student's current state. In other words, we partition the space of programs according to several basins of attraction, one per correct solution. The setup and training remain the same, otherwise.

This concludes our description of methods. In the next section, we evaluate these methods on large-scale datasets.

4. Evaluation

In this section, we evaluate the ast2vec model on two large-scale anonymized datasets of introductory programming, namely the 2018 and 2019 beginners challenge by the National Computer Science School (NCSS)[5], an Australian educational outreach program. The courses were delivered by the Grok Learning platform[6]. Each offering was available to (mostly) Australian school children in Years 5–10, with curriculum-aligned educational slides and sets of exercises released each week for five weeks. Students received a score for successfully completing each exercise, with the score available starting at 10 points, reducing by one point every five incorrect submissions, to a minimum of 5 points. In both datasets, we consider only submissions, i.e., programs that students deliberately submitted for evaluation against test cases.

The 2018 dataset contains data of $12,141$ students working on 26 different programming tasks, yielding $148,658$ compilable programs. This is also part of the data on which ast2vec was trained. The 2019 dataset contains data of $10,558$ students working on 24 problems, yielding $194,797$ compilable programs overall.

For our first analysis, we further broaden our scope and include a third dataset with a slightly different course format, in particular, the Australian Computing Academy digital technologies (DT) chatbot project[7] which consists of 63 problems and teaches students the skills to program

[5] https://ncss.edu.au
[6] https://groklearning.com/challenge/
[7] https://aca.edu.au/resources/python-chatbot/

a simple chatbot and is also delivered via the Grok Learning platform. Our dataset includes the data of $27,480$ students enrolled between May 2017 and August 2020, yielding $1,343,608$ compilable programs.

We first check how well ast2vec is able to correctly autoencode trees in all three datasets and then go on to check its utility for prediction. We close the section by inspecting the coding space in more detail for the example dataset from Section 2.

4.1. Autoencoding error

Our first evaluation concerns the ability of ast2vec to maintain information in its encoding. In particular, we evaluate the autoencoding error, i.e., we iterate over all compilable programs in the dataset, compile them to an abstract syntax tree (AST), use ast2vec to convert the AST into a vector, decode the vector back into an AST, and compute the tree edit distance (Zhang and Shasha, 1989) to the original AST.

Figure 8 shows the autoencoding error for trees of different sizes in the three datasets. The black line shows the median autoencoding error, whereas the orange line shows the average, with the orange region indicating one standard deviation. The blue dashed line indicates the cumulative distribution, i.e., the fraction of trees in the data that have at most x nodes.

With respect to the 2018 data, we observe that the median is generally lower than the mean, indicating that the distribution has a long tail of few trees with higher autoencoding error, whereas most trees have low autoencoding error. Indeed, up to tree size 36, the median autoencoding error is zero, and 78.92% of all programs in the dataset are smaller than 37 nodes, thus covering a sizeable portion of the dataset. The overall average autoencoding error across the dataset is 1.86 and the average tree size is 27.98.

With respect to the 2019 dataset, we observe that median and average are more closely aligned, indicating more symmetric distributions of errors. Additionally, we notice that the autoencoding error grows more quickly compared to the 2018 data, which indicates that ast2vec is slightly worse in autoencoding programs outside its training data compared to programs inside its training data. We obtain an average of 4.10 compared to an average tree size of 24.28.

Similarly, for the chatbot data, we observe the average overtaking the median and that the error is generally larger compared to the 2018 data. A specific property of the chatbot dataset is that there is a long tail of large programs which correspond to the full chatbot. If we include these in the analysis, we obtain an average autoencoding error of 7.86 and an average tree size of 27.40. If we only consider trees up to size 85, covering 95% of all trees in the dataset, we obtain an average autoencoding error of 4.41 and an average tree size of 22.49.

Overall, we observe that ast2vec performs better on smaller trees and better on trees in the training data. This is not surprising but should be taken into account when applying ast2vec to new datasets. One may also be able to further reduce the autoencoding error by adjusting the ast2vec architecture, e.g., by incorporating more recurrent nets such as Tree LSTMs (Chen et al., 2018; Dai et al., 2018; Tai et al., 2015).

Figure 9 shows the time needed to encode (orange) and decode (blue) a tree from the chatbot dataset of a given size. As the plot indicates, the empirical time complexity for both operations is roughly linear in the tree size, which corresponds to the theoretical findings of Paaßen et al. (2021). Using a linear regression without intercept, we find that encoding requires roughly 0.9 milliseconds per ten tree nodes, whereas decoding requires roughly 2.8 milliseconds per ten tree nodes. Given that decoding involves more operations (both element choice and vector

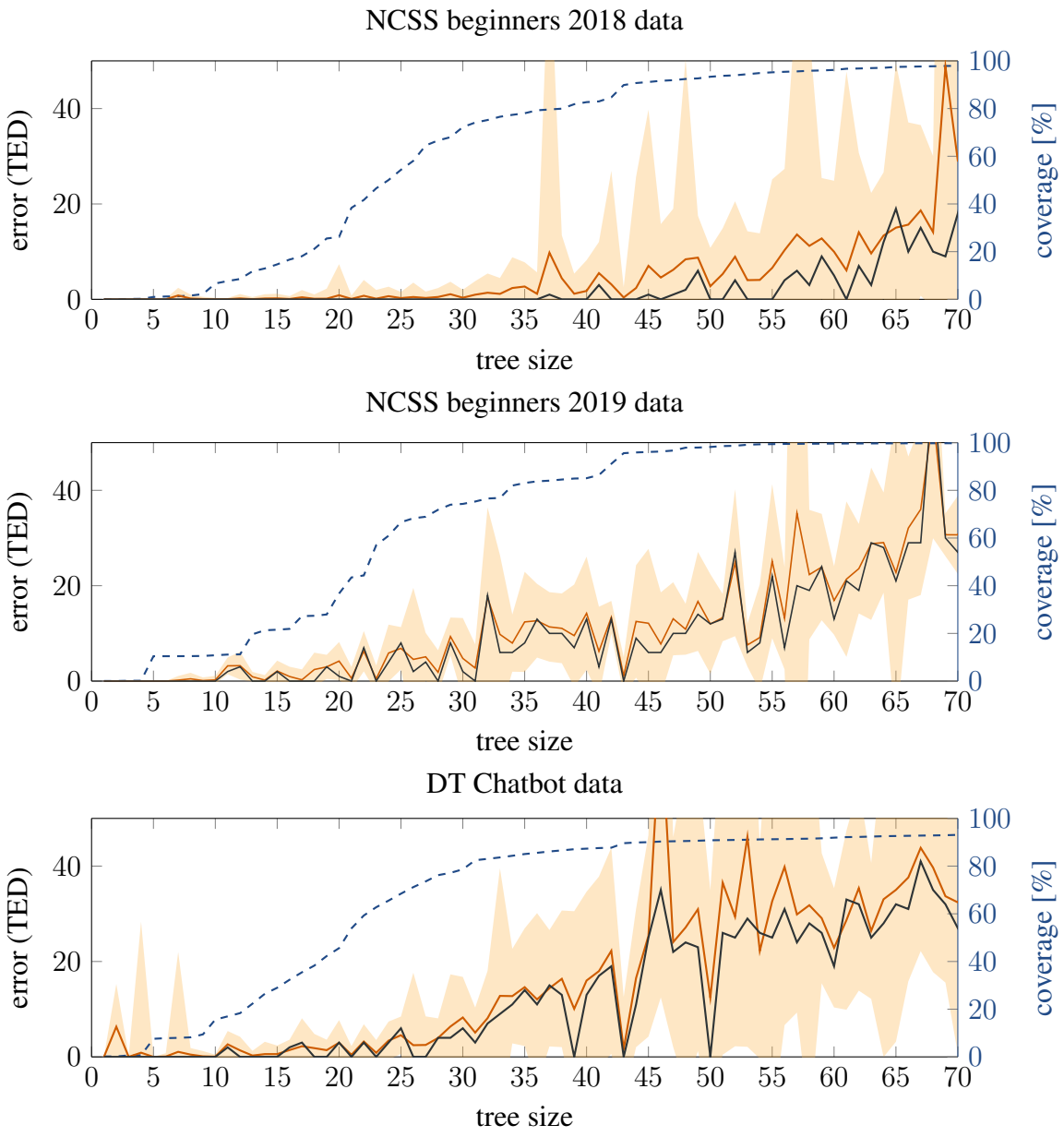

Figure 8: The autoencoding error as measured by the tree edit distance for the NCSS 2018 beginners challenge (top), NCSS 2019 beginner challenge (middle), and DT chatbot course (bottom). The error is plotted here versus tree size. The orange line marks the mean, the black line the median error. The orange region is the standard deviation around the man. Additionally, the blue line indicates how many trees in the dataset have a size up to x.

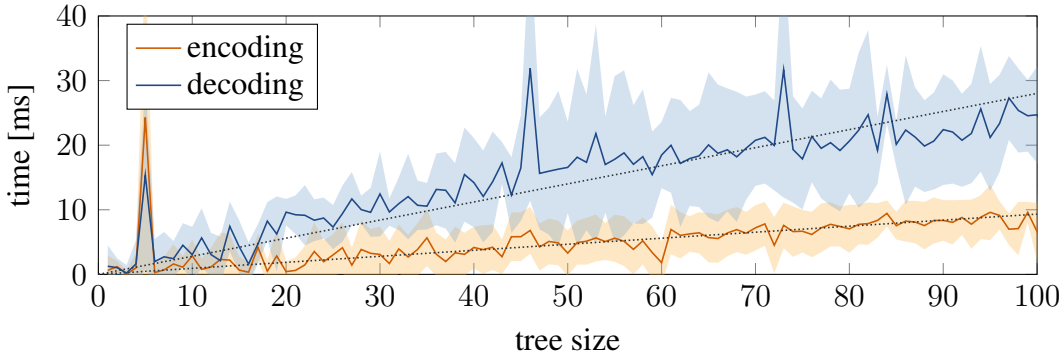

Figure 9: The time needed for encoding (orange) and decoding (blue) trees of different sizes for the DT chatbot course. The thick lines mark the means, whereas shaded regions indicate one standard deviation. The dotted lines indicate the best linear fit.

operations), this difference in runtime is to be expected. Fortunately, both operations remain fast, even for relatively large trees.

4.2. Next-Step Prediction

In the previous section, we evaluated how well ast2vec could translate from programs to vectors and back, finding that it did well on the majority of beginner programs. In this section, we investigate the dynamical systems analysis suggested in Sections 2.3 and 3.3. In particular, we compare its ability to predict the next step of a student based on example data from other students.

Recall that our dynamical systems model is quite simple: It encodes the student's current abstract syntax tree as a vector, computes the difference in the encoding space to the most common correct solution, applies a linear transformation to that difference, adds the result to the student's current position, and decodes the resulting vector to achieve the next-step-prediction. Given this simplicity, our research objective in this section is not to provide the best possible next-step prediction. Rather, we wish to investigate how well a very simple model can perform just by virtue of using the continuous vector space of ast2vec.

By contrast, we compare to the following reference models from the literature:

1. the simple identity, which predicts the next step to be the same as the current one and is included as a baseline (Hyndman and Koehler, 2006),

2. one-nearest-neighbor prediction (1NN). This involves searching the training data for the closest tree based on tree edit distance and predicting its successor, which is similar to the scheme suggested by Gross and Pinkwart (2015), and

3. the continuous hint factory (Paaßen et al., 2018, CHF), which is also based on the tree edit distance but instead involves Gaussian process regression and heuristic-driven reconstruction techniques to predict next steps.

Both 1NN as well as CHF are nonlinear predictors with a much higher representational capacity, such that we would expect them to be more accurate (Paaßen et al., 2018). We investigate this in the following section.

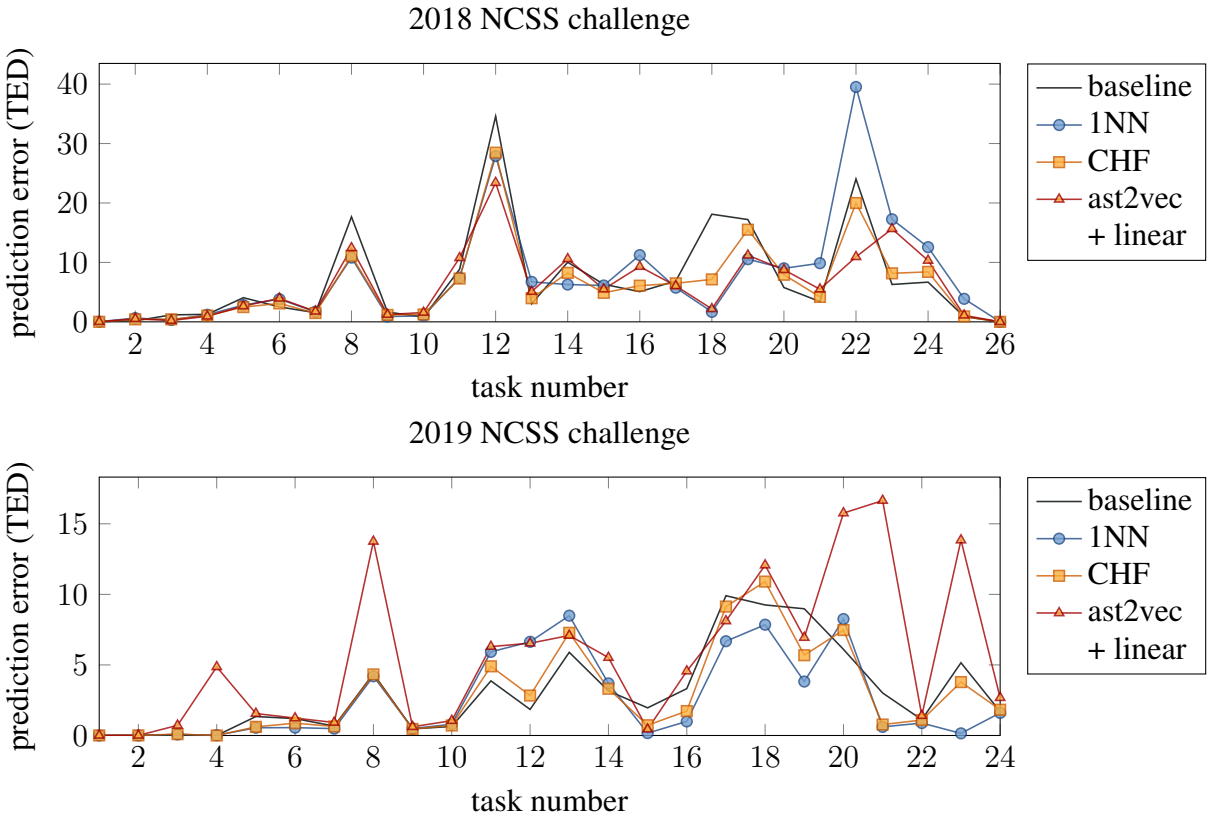

Figure 10: The average root mean square prediction error (in terms of tree edit distance) across the entire beginners 2018 (top) and 2019 (bottom) curriculum for different prediction methods. The x-axis uses the same order of problems as students worked on them.

4.2.1. Predictive error

We evaluate the error in next-step prediction on two different datasets: the first was the 2018 NCSS challenge data, which was used to train ast2vec, and the second was the 2019 NCSS challenge data, which contained programs unseen by ast2vec. We considered each learning task in both challenges separately. For each task, we partitioned the student data into 10 folds and performed 10-fold cross-validation, i.e., using nine folds as training data and one fold as evaluation data. Additionally, we subsampled the training data to only contain the data of 30 students in order to simulate a classroom-sized training dataset.

Figure 10 shows the average root mean square prediction error in terms of tree edit distance on the 2018 and 2019 data. In general, our simple linear model performed comparably to the other models, with the performance being particularly close for the 2018 data. This difference between the datasets is to be expected, given that ast2vec has lower autoencoding error on the 2018 dataset.

With respect to the 2019 data, our model still performed reasonably well for most exercises, though there were some cases where the error was noticeably higher (i.e., tasks 4, 8, 20, 21, and 23). The error on tasks 4, 20, 21, and 23 can be explained by the fact that these tasks involved larger programs (task 4 was an early exercise, but it involved a large program scaffold that students needed to modify). Based on the analysis in the previous section, larger programs tend

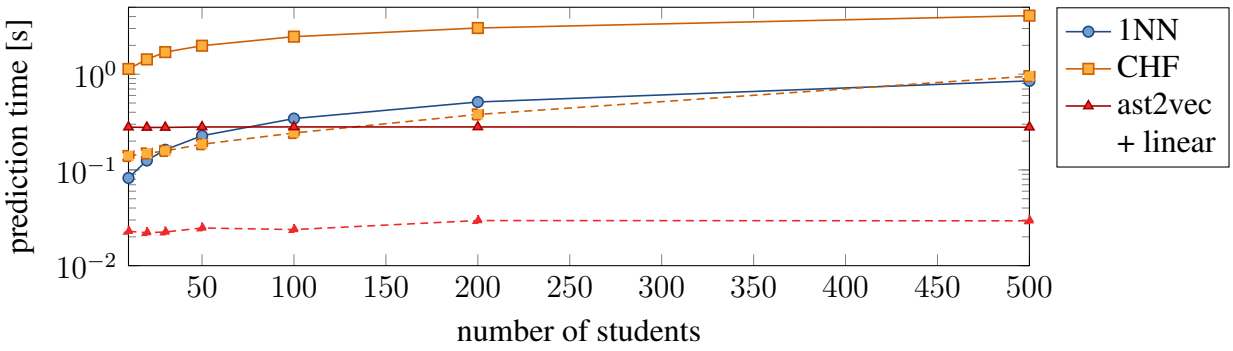

Figure 11: The average training (dashed lines) and prediction time (solid lines) for different training set sizes for the 2019 dataset. Prediction times are measured as the accumulated time to provide predictions for an entire student trace. Note that the y axis is in log scale.

to have higher autoencoding error, which we would expect to affect the prediction error. Task 8, however, is interesting because it involves relatively small programs. On closer inspection, this task has many different possible solutions, so the error could be related to the simple design of our model (specifically, that it only makes predictions towards the most common solution). All-in-all, this suggests the simple model performs comparably to the others and only performs worse in a small number of specific and explainable cases.

We note in passing that the naive baseline of predicting the current step also performs quite well in both datasets. This may seem surprising but is a usual finding in forecasting settings (Hyndman and Koehler, 2006; Paaßen et al., 2018). In particular, a low baseline error merely indicates that students tend to change their program only a little in each step, which is to be expected. Importantly, though, the baseline can not be used in an actual hint system, where one would utilize the difference between a student's current state and the predicted next state to generate hints (Rivers and Koedinger, 2017; Paaßen et al., 2018). For the baseline, there is no difference, and hence no hints can be generated.

4.2.2. Runtime

Another interesting aspect is training and prediction time. While training is very fast across all methods (1NN does not even need training), the prediction time of neighborhood-based methods like 1NN and CHF scales with the size of the training dataset, whereas the prediction time for our ast2vec+linear scheme remains constant. Figure 11 displays the average training and prediction times for all methods and varying training dataset sizes on the 2019 dataset. As we can see, our proposed scheme always needs about 300ms to make all predictions for a student trace (including encoding and decoding times), whereas 1NN exceeds this time starting from ≈ 80 students, and CHF is slower across time scales, requiring more than a second even for small training data sizes. Prediction time is an important consideration in large-scale educational contexts, where many predictions may need to be made very quickly.

4.2.3. Qualitative Comparison

Beyond the quantitative comparison in terms of prediction error and runtime, there are important qualitative differences between our proposed dynamical systems and previous neighborhood-

based schemes. As noted in Section 3.3, one important property of our simple model is that the predictions are mathematically guaranteed to lead to a solution, and the linearity of the model means this is done smoothly. By contrast, neither 1NN nor CHF can formally guarantee convergence to a correct solution if one follows the predictions. Additionally, the predictions of 1NN are discontinuous, i.e., they change suddenly if the student enters a different neighborhood. Overall, a smooth and guaranteed motion towards a correct solution could be valuable for designing a next-step hint system where the generated hints should be both directed to the desired target and intuitive (McBroom et al., 2021; Paaßen et al., 2018; Rivers and Koedinger, 2017).

Furthermore, a dynamical system based on ast2vec only has to solve the problem of describing student motion in the space of programs for a particular task. Constructing the space of programs in the first place is already solved by ast2vec. By contrast, a neighborhood-based model must solve both the representation *and* the learning problem anew for each learning task.

Additionally, any neighborhood-based model must store the entire training dataset at all times and recall it for every prediction, whereas the ast2vec+linear model merely needs to store and recover the predictive parameters, which includes the ast2vec parameters, and for each learning task, the vector encoding of the correct solution and the matrix $\boldsymbol{W}$. This is not only more time-efficient (as we saw above) but also space efficient as each new learning task only requires an additional $n + 1 \times n = 65,792$ floating point numbers to be stored, which is roughly half a megabyte.

Finally, if additional predictive accuracy is desired, one can improve the predictive model by replacing the linear prediction with a nonlinear prediction (e.g., a small neural network), whereas the predictive accuracy of a neighborhood-based system can only be improved by using a richer distance measure or more training data.

In summary, this evaluation has shown that a simple linear regression model using the vector encodings of ast2vec can perform comparably to neighborhood-based next-step prediction techniques in terms of prediction error but has other desirable properties that may make it preferable in many settings, such as constant time and space complexity, guaranteed convergence to a correct solution, smoothness, and separation of concerns between representation and prediction. As such, we hope that ast2vec can contribute to improving educational data mining pipelines, e.g., for next-step hint predictions in the future.

4.3. Interpolation in the coding space

In the previous sections, we have established that ast2vec is generally able to decode vectors back to fitting syntax trees. However, what remains open is whether the coding space is *smooth*, i.e., whether neighboring points in the coding space correspond to similar programs. This property is important to ensure that typical analyses like visualization and dynamical systems from Section 2 are meaningful. More specifically, if we look at a visualization of a dataset, we implicitly assume that points close in the plot also correspond to similar programs. Similarly, when we perform dynamical system analysis, we implicitly assume that small movements in the $n = 256$-dimensional coding space also correspond to small changes in the corresponding program.

In principle, the variational autoencoder framework (refer to Section 3) ensures such a smoothness property. In this section, we wish to validate this for an example. In particular, Figure 12 shows a 2D grid of points which we sample from the progress-variance projection for our example dataset from Section 2. Points that decode to the same syntax tree receive the

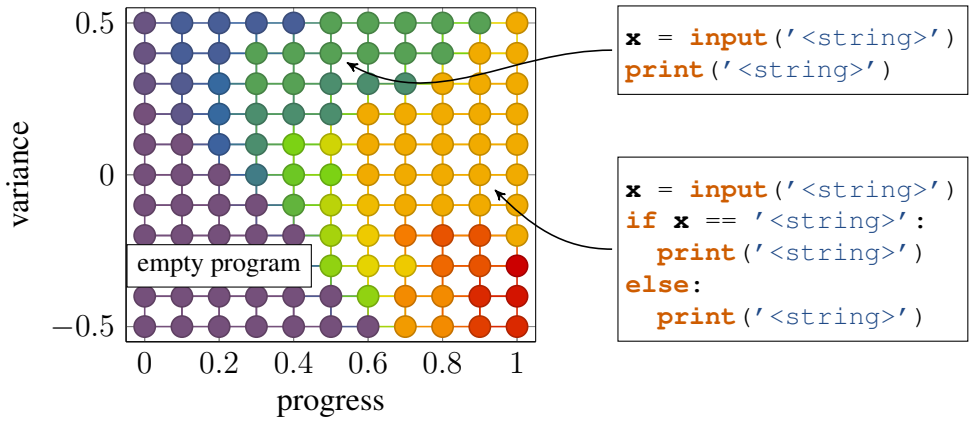

Figure 12: A linear interpolation between the empty program at $(0, 0)$ and the correct solution at $(1, 0)$ as well as along the axis of greatest variation. The color indicates the tree that the grid point decodes to. The three most common trees are shown in the boxes.

same color. The programs corresponding to regions with at least ten grid points are shown on the right.

This visualization shows several reassuring properties: First, neighboring points in the grid tend to decode to the same tree. Second, if two points correspond to the same tree, the space between them also correspond to the same tree, i.e., there are no two points corresponding to the same tree at completely disparate locations of the grid. Finally, the syntax trees that are particularly common are also meaningful for the task in the sense that they actually occur in student data. Overall, this bolsters our confidence that the encoding space is indeed smooth.

5. Related Work

In this paper, we propose a novel, vectorial encoding for computer programs to support educational data mining. In doing so, we build on prior work, which also proposed alternative representations for computer programs. In the remainder of this section, we review these alternative representations and relate them to ast2vec.

Abstract Syntax Trees. An abstract syntax tree (AST) represents the program code as a tree of syntactic elements, which is also the internal representation used by compilers to transform human-readable program code to machine code (Aho et al., 2006). For example, Figure 5 (left) shows the abstract syntax tree for the Python program `print('Hello, world!')`. In most processing pipelines for computer programs, compiling the source code into an AST is the first step, because it tokenizes the sources code meaningfully, establishes the hierarchical structure of the program, and separates functionally relevant syntax from functionally irrelevant syntax, like variable names or comments (Aho et al., 2006; McBroom et al., 2021). Additionally, several augmentations have been suggested to make syntax trees more meaningful. In particular, Rivers and Koedinger (2012) suggested canonicalization rules to match trees that are functionally equivalent, such as normalizing the direction of comparison operators, inlining helper functions, or removing unreachable code. Gross et al. (2014) further suggested inserting edges in

the tree between variable names and their original declaration, thus augmenting the tree to a graph. Following prior work, we also use the AST representation as a first step before we apply a neural network to it. Although we do not use them here explicitly, the canonicalization rules of Rivers and Koedinger (2012) are directly compatible with ast2vec. The additional reference edges suggested by Gross et al. (2014) are less compatible because these would require neural networks that can deal with full graphs, i.e., graph neural networks (Kipf and Welling, 2016; Scarselli et al., 2009).

TREE PATTERNS. Since many data mining methods can not be applied to trees, several prior works have suggested transforming the AST to a collection of patterns first. Such representations are common in the domain of natural language processing, where texts can be represented as a collection of the words or n-grams they contain (Broder et al., 1997). Such techniques can be extended to trees by representing a tree by the subtree patterns it contains (Nguyen et al., 2014; Zhi et al., 2018; Zimmerman and Rupakheti, 2015). For example, the syntax tree "Module(Expr(Call(Name, Str)))" could be represented as the following collection of subtree patterns of height two: "Call(Name, Str)", "Expr(Call)", and "Module(Expr)". Once this collection is computed, we can associate all possible subtree patterns with a unique index in the range $1, \ldots, K$ and then represent a tree by a K-dimensional vector, where entry k counts how often the kth subtree pattern occurs in the tree.

If the collection of subtree patterns is meaningful for the programming task at hand, this representation can be both simple and powerful. For example, if a programming task is about writing a while loop, it is valuable to know whether a student's program actually contains such a loop. Zhi et al. (2018) have considered this issue in more detail and considered both expert-based and data-driven tree patterns as a representation. However, if there is no clear relation between tree patterns and progress in a task, it may be problematic to use such pattern as a representation, in particular when all tree patterns are weighted equally. Ast2vec is weakly related to tree patterns because the tree pattern determines which encoding/decoding functions are called when processing the tree. However, the vector returned by the network does not simply count tree patterns but, instead, considers the entire tree structure, i.e., how the tree patterns are nested into each other to form the entire tree. More precisely, ast2vec is trained to generate vector encodings that still contain enough information to recover the original tree, whereas tree pattern counting usually does not enable us to recover the original tree.

PAIRWISE DISTANCES. An alternative to an explicit vector representation is offered by pairwise distances (or similarities). Most prominently, abstract syntax trees can be represented by their pairwise tree edit distance $d(x, y)$, which is defined as the number of syntactic elements one has to delete, insert, or re-label to get from tree x to tree y (Mokbel et al., 2013; Paaßen et al., 2018; Price et al., 2017; Rivers and Koedinger, 2017; Zhang and Shasha, 1989). Such a pairwise distance representation is mathematically quite rich. Indeed, one can prove that a distance representation implicitly assigns vectors to programs and one can facilitate this implicit representation for distance-based variants of machine learning methods, e.g., for visualization, classification, and clustering (Pekalska and Duin, 2005; Hammer and Hasenfuss, 2010). Additionally, an edit distance enables us to infer the changes we have to apply to a tree $\hat{x}$ to get to another tree $\hat{y}$, which facilitates hints (Paaßen et al., 2018; Price et al., 2017; Rivers and Koedinger, 2017). However, there are several subtle problems that can complicate a practical application. First, there may exist exponentially many edit paths between $\hat{x}$ and $\hat{y}$, and it is

not always easy to choose among them, especially since many intermediate trees may be syntactically invalid or implausible to students (Paaßen et al., 2018; Rivers and Koedinger, 2017). More generally, distances require reference programs to which distances can be computed. This makes distance-based methods challenging to scale for larger datasets where many reference programs exist. By contrast, ast2vec can translate a new program into a vector without reference to past data. Only the model parameters have to be stored, which is constant size and runtime versus linear size and runtime. Finally, ast2vec enables us to decouple the general problem of representing programs as vectors from task-specific problems. We can utilize the implicit information of 500,000 student programs in ast2vec in a new task and may need only a little new student data to solve the additional, task-specific problem at hand.

That being said, the basic logic of distance measures is still crucial to ast2vec. In particular, the negative log-likelihood in Equation 9 can be interpreted as a measure of distance between the original program and its autoencoded version, and our notion of a smooth encoding can be interpreted as small distances between vectors in the encoding space implying small distances of the corresponding programs.

CLUSTERING. One of the challenges of computer programs is that the space of programs for the same task is very large, and it is infeasible to design feedback for all possible programs. Accordingly, several researchers have instead grouped programs into a small number of clusters, for which similar feedback can be given (Choudhury et al., 2016; Glassman et al., 2015; Gross et al., 2014; Gulwani et al., 2018). Whenever a new student requests help, one can simply check which cluster the student's program belongs to and assign the feedback for that cluster. One typical way to perform clustering is by using a pairwise distance measure on trees, such as the tree edit distance (Choudhury et al., 2016; Gross et al., 2014; Zhang and Shasha, 1989). However, it is also possible to use clustering strategies more specific to computer programs, such as grouping programs by their control flow (Gulwani et al., 2018) or by the unit tests they pass (McBroom et al., 2021). ast2vec can be seen as a preprocessing step for clustering. Once all programs are encoded as vectors, standard clustering approaches can be applied. Additionally, ast2vec provides the benefit of being able to interpret the clustering by translating cluster centers back into syntax trees (refer to Section 2.4).

EXECUTION BEHAVIOR. Most representations so far focused on the AST of a program. However, it is also possible to represent programs in terms of their execution behavior. The most popular example is the representation by test cases, where a program is fed with example input. If the program's output is equal to the expected output, we say the test has passed; otherwise, it has failed (Ihantola et al., 2010). This is particularly useful for automatic grading of programs because test cases verify the functionality, at least for some examples, while giving students freedom on how to implement this functionality (Ihantola et al., 2010). Further, failing a certain unit test can indicate a specific misconception, warranting functionality-based feedback (McBroom et al., 2021). However, for certain types of tasks, the computational path toward an output may be relevant, for example, when comparing sorting programs. Paaßen et al. (2016) therefore use the entire computation trace as a representation, i.e., the sequence of states of all variables in a program. Unfortunately, a mismatch in the output or in the execution behavior toward the output is, in general, difficult to relate to a specific change the student would need to perform in order to correct the problem. To alleviate this challenge, test cases have to be carefully and densely designed, or the challenge has to be left to the student. As such, we believe that there is

still ample room for AST-based representations, like our proposed vector encodings, which are closer to the actions a student can actually perform on their own code.

NEURAL NETWORKS. Prior work already has investigated neural networks to encode computer programs. In particular, Piech et al. (2015) used a recursive tensor network to encode syntax trees and performed classification on the embeddings to decide whether certain feedback should be applied or not; Alon et al. (2019) represented syntax trees as a collection of paths, encoded these paths as vectors, and then computed a weighted average to obtain an overall encoding of the program; Chen et al. (2018) translated programs into other programming languages by means of an intermediary vector representation; and Dai et al. (2018) proposed an autoencoding model for computer programs that incorporates an attribute grammar of the programming language to not only incorporate syntactic but also semantic constraints (e.g., that variables can only be used after declaration).

Both the works of Piech et al. (2015) and Alon et al. (2019) are different from our contribution because they do not optimize an autoencoding loss but instead train a neural net for the classification of programs, i.e., into feedback classes or into tags. This scheme is unable to recover the original program from the network's output and requires expert labeling for all training programs. Ast2vec has neither of those limitations. The work of Chen et al. (2018) is more similar in that both input and output are programs. However, the network does not include grammar constraints and uses an attention mechanism on the original tree to decide on its output. This is not possible in our setting where we wish to perform data mining solely in the vector space of encodings and be able to decode any encoding back into a tree without reference to a prior tree. The most similar prior work to our own is the autoencoding model of Dai et al. (2018), which is also an autoencoding model with grammar constraints, albeit with an LSTM-based encoder and decoder. One could frame ast2vec as a combination of the autoencoding ability and grammar knowledge of Dai et al. (2018) with the recursive processing of Piech et al. (2015), yielding a recursive tree grammar autoencoder (Paaßen et al., 2021). That being said, future work may incorporate more recurrent network concepts and thus improve autoencoding error further.

NEXT-STEP HINTS. Ample prior work has considered the problem of predicting the next step a student should take in a learning system, refer e.g., to the review of McBroom et al. (2021). On a high level, this concerns the problem of selecting the right sequence of lessons to maximize learning gain (Lan et al., 2014; Reddy et al., 2016). In this paper, we rather consider the problem of predicting the next code change within a single programming task. This problem has been considered in more detail, for example, by Rivers and Koedinger (2017), Price et al. (2017), Paaßen et al. (2018), and Price et al. (2019). Here, we compare to two baselines evaluated by Price et al. (2019), namely the continuous hint factory (Paaßen et al., 2018) and a nearest-neighbor prediction (Gross and Pinkwart, 2015). We note, however, that we only evaluate the predictive accuracy, not the hint quality, which requires further analysis (Price et al., 2019).

6. CONCLUSION

In this paper, we presented ast2vec, a novel autoencoder to translate the syntax trees of beginner Python programs to vectors in Euclidean space and back. Such a vectorial representation is useful because it enables at least a hundred data mining techniques (judging from the sklearn package in Python) that would not be possible without a vectorial representation.

We have trained our autoencoder on almost half a million beginner Python programs and evaluated it in three settings. First, we utilized the network for a variety of analyses on a classroom-sized dataset, illustrating its broad applicability for making sense of student data and supporting educational decisions. As part of our qualitative analysis, we also showed that the encoding space of ast2vec is smooth - at least for the example - and introduced two novel techniques for analyzing programming data based on ast2vec, namely progress-variance-projections for a two-dimensional visualization and a linear dynamical systems method that guarantees convergence to the correct solution.

In terms of quantitative analyses, we evaluated the autoencoding error of ast2vec, the predictive accuracy of a simple linear model on top of ast2vec, and the runtime of both. We found that ast2vec had a low autoencoding error for the majority of programs, including on two unseen, large-scale datasets, though the error tended to increase with tree size. In addition, the encoding and decoding times were linear in the tree size with low scaling factor, suggesting ast2vec is scalable. Moreover, by coupling ast2vec with our linear dynamical systems method, we were able to approach the predictive accuracy of existing methods with a very simple model that is guaranteed to converge to a correct solution and requires only constant time and space, whereas existing baselines require a database with linear time and space constraints.

While we believe that these results are encouraging, we also acknowledge limitations: At present, ast2vec does not decode the content of variables, although such content may be decisive to solve a task correctly. Further improvements in terms of autoencoding error are possible as well, e.g., by including more advanced recurrent networks as encoders and decoders. Finally, the predictive accuracy of our proposed linear dynamical systems model has not yet achieved the state-of-the-art and nonlinear predictors are likely necessary to improve performance further.

Still, ast2vec provides the educational data mining community with a novel tool that can be utilized without any need for further deep learning in a broad variety of analyses. We are excited to see its applications in the future.

7. Acknowledgments

Funding by the German Research Foundation (DFG) under grant numbers PA 3460/1-1 and PA 3460/2-1 is gratefully acknowledged.

8. Editorial Statement

Kalina Yacef had no involvement with the journal's handling of this article in order to avoid a conflict with her Associate Editor role. The entire review process was managed by Special Guest Editors Olga C. Santos and Amal Zouaq.

References

Aggarwal, C. C., Hinneburg, A., and Keim, D. A. 2001. On the surprising behavior of distance metrics in high dimensional space. In *Proceedings of the International Conference on Database Theory (ICDT 2001)*, J. Van den Bussche and V. Vianu, Eds. Springer, Berlin, Heidelberg, 420–434.

Aho, A., Lam, M., Sethi, R., and Ullman, J. 2006. *Compilers: Principles, Techniques, and Tools*, 2 ed. Addison Wesley, Boston, MA, USA.

ALON, U., ZILBERSTEIN, M., LEVY, O., AND YAHAV, E. 2019. Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages 3*, 40.

BAKER, R. S. AND YACEF, K. 2009. The state of educational data mining in 2009: A review and future visions. *Journal of Educational Data Mining 1,* 1, 3–17.

BARNES, T., MOSTAFAVI, B., AND EAGLE, M. J. 2016. Data-driven domain models for problem solving. In *Design Recommendations for Intelligent Tutoring Systems*, R. Sottilare, A. Graesser, X. Hu, A. Olney, B. Nye, and A. Sinatra, Eds. Vol. 4. US Army Research Laboratory, Orlando, FL, USA, 137–145.

BRODER, A. Z., GLASSMAN, S. C., MANASSE, M. S., AND ZWEIG, G. 1997. Syntactic clustering of the web. *Computer Networks and ISDN Systems 29,* 8, 1157 – 1166.

CAMPI, M. AND KUMAR, P. 1998. Learning dynamical systems in a stationary environment. *Systems & Control Letters 34,* 3, 125 – 132.

CHEN, X., LIU, C., AND SONG, D. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the 31st International Conference on Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates Inc., 2552–2562.

CHO, K., VAN MERRIENBOER, B., GÜLÇEHRE, Ç., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, A. Moschitti, B. Pang, and W. Daelemans, Eds. 1724–1734.

CHOUDHURY, R. R., YIN, H., MOGHADAM, J., AND FOX, A. 2016. Autostyle: Toward coding style feedback at scale. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, D. Gergle, M. R. Morris, P. Bjørn, J. Konstan, G. Hsieh, and N. Yamashita, Eds. 21–24.

DAI, H., TIAN, Y., DAI, B., SKIENA, S., AND SONG, L. 2018. Syntax-directed variational autoencoder for structured data. In *Proceedings of the 6th International Conference on Learning Representations*, Y. Bengio, Y. LeCun, T. Sainath, I. Murray, M. A. Ranzato, and O. Vinyals, Eds.

DEMPSTER, A. P., LAIRD, N. M., AND RUBIN, D. B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological) 39,* 1, 1–38.

DENNING, P. J. 2017. Remaining trouble spots with computational thinking. *Communications of the ACM 60,* 6, 33–39.

GISBRECHT, A. AND SCHLEIF, F.-M. 2015. Metric and non-metric proximity transformations at linear costs. *Neurocomputing 167*, 643–657.

GLASSMAN, E. L., SCOTT, J., SINGH, R., GUO, P. J., AND MILLER, R. C. 2015. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction 22,* 2, 7.

GROSS, S., MOKBEL, B., PAASSEN, B., HAMMER, B., AND PINKWART, N. 2014. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 9,* 3, 248–280.

GROSS, S. AND PINKWART, N. 2015. How do learners behave in help-seeking when given a choice? In *Proceedings of the 17th International Conference on Artificial Intelligence in Education*, C. Conati, N. Heffernan, A. Mitrovic, and M. F. Verdejo, Eds. 600–603.

GULWANI, S., RADIČEK, I., AND ZULEGER, F. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices 53,* 4, 465–480.

HAMMER, B. AND HASENFUSS, A. 2010. Topographic mapping of large dissimilarity data sets. *Neural Computation 22,* 9, 2229–2284.

HAZAN, E., SINGH, K., AND ZHANG, C. 2017. Learning linear dynamical systems via spectral filtering. In *Proceedings of the 30th Conference on Advances Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. 6702–6712.

HYNDMAN, R. J. AND KOEHLER, A. B. 2006. Another look at measures of forecast accuracy. *International Journal of Forecasting 22,* 4, 679 – 688.

IHANTOLA, P., AHONIEMI, T., KARAVIRTA, V., AND SEPPÄLÄ, O. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, C. Schulte and J. Suhonen, Eds. 86–93.

KINGMA, D. AND BA, J. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, Y. Bengio, Y. LeCun, B. Kingsbury, S. Bengio, N. de Freitas, and H. Larochelle, Eds.

KINGMA, D. AND WELLING, M. 2013. Auto-encoding variational Bayes. In *Proceedings of the 1st International Conference on Learning Representations*, Y. Bengio, Y. LeCun, A. Courville, R. Fergus, and C. Manning, Eds.

KIPF, T. N. AND WELLING, M. 2016. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 4th International Conference on Learning Representations*, H. Larochelle, S. Bengio, B. Kingsbury, Y. Bengio, and Y. LeCun, Eds.

KOPRINSKA, I., STRETTON, J., AND YACEF, K. 2015. Predicting student performance from multiple data sources. In *Proceedings of the International Conference on Artificial Intelligence in Education*, C. Conati, N. Heffernan, A. Mitrovic, and M. Verdejo, Eds. 678–681.

KUSNER, M. J., PAIGE, B., AND HERNÁNDEZ-LOBATO, J. M. 2017. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds. 1945–1954.

LAHTINEN, E., ALA-MUTKA, K., AND JÄRVINEN, H.-M. 2005. A study of the difficulties of novice programmers. *SIGCSE Bulletin 37,* 3, 14–18.

LAN, A. S., STUDER, C., AND BARANIUK, R. G. 2014. Time-varying learning and content analytics via sparse factor analysis. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, S. Macskassy, C. Perlich, J. Leskovec, W. Wang, and R. Ghani, Eds. 452–461.

MCBROOM, J., KOPRINSKA, I., AND YACEF, K. 2020. How does student behaviour change approaching dropout? A study of gender and school year differences. In *Proceedings of the 13th International Conference on Educational Data Mining*, A. Rafferty, J. Whitehill, V. Cavalli-Sforza, and C. Romero, Eds. 643–647.

MCBROOM, J., KOPRINSKA, I., AND YACEF, K. 2021. A survey of automated programming hint generation: The hints framework. *ACM Computing Surveys 54,* 8, 172.

MCBROOM, J., PAASSEN, B., JEFFRIES, B., KOPRINSKA, I., AND YACEF, K. 2021. Progress networks as a tool for analysing student programming difficulties. In *Proceedings of the Twenty-Third Australasian Conference on Computing Education*, C. Szabo and J. Sheard, Eds. 158–167.

MCBROOM, J., YACEF, K., KOPRINSKA, I., AND CURRAN, J. R. 2018. A data-driven method for helping teachers improve feedback in computer programming automated tutors. In *Artificial Intelligence in Education*, C. Penstein Rosé, R. Martínez-Maldonado, H. U. Hoppe, R. Luckin, M. Mavrikis, K. Porayska-Pomsta, B. McLaren, and B. du Boulay, Eds. 324–337.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. 125–180.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. 3111–3119.

Mokbel, B., Gross, S., Paassen, B., Pinkwart, N., and Hammer, B. 2013. Domain-independent proximity measures in intelligent tutoring systems. In *Proceedings of the 6th International Conference on Educational Data Mining*, S. K. D'Mello, R. A. Calvo, and A. Olney, Eds. 334–335.

Nguyen, A., Piech, C., Huang, J., and Guibas, L. 2014. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, C.-W. Chung, A. Broder, K. Shim, and T. Suel, Eds. Association for Computing Machinery, 491–502.

Paassen, B., Göpfert, C., and Hammer, B. 2018. Time series prediction for graphs in kernel and dissimilarity spaces. *Neural Processing Letters 48,* 2, 669–689.

Paassen, B., Hammer, B., Price, T., Barnes, T., Gross, S., and Pinkwart, N. 2018. The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces. *Journal of Educational Data Mining 10,* 1, 1–35.

Paassen, B., Jensen, J., and Hammer, B. 2016. Execution traces as a powerful data representation for intelligent tutoring systems for programming. In *Proceedings of the 9th International Conference on Educational Data Mining*, T. Barnes, M. Chi, and M. Feng, Eds. International Educational Data Mining Society, 183 – 190.

Paassen, B., Koprinska, I., and Yacef, K. 2021. Recursive tree grammar autoencoders. *arXiv 2012.02097*. https://arxiv.org/abs/2012.02097.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 32nd International Conference on Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. 8026–8037.

Pearson, K. 1901. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2,* 11, 559–572.

Peddycord, III, B., Hicks, A., and Barnes, T. 2014. Generating hints for programming problems using intermediate output. In *Proceedings of the 7th International Conference on Educational Data Mining*, M. Mavrikis and B. M. McLaren, Eds. 92–98.

Pekalska, E. and Duin, R. 2005. *The Dissimilarity Representation for Pattern Recognition: Foundations And Applications (Machine Perception and Artificial Intelligence)*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

Pennington, J., Socher, R., and Manning, C. D. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, A. Moschitti, B. Pang, and W. Daelemans, Eds. 1532–1543.

PIECH, C., HUANG, J., NGUYEN, A., PHULSUKSOMBATI, M., SAHAMI, M., AND GUIBAS, L. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 37th International Conference on Machine Learning*, F. Bach and D. Blei, Eds. 1093–1102.

PIECH, C., SAHAMI, M., HUANG, J., AND GUIBAS, L. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second ACM Conference on Learning @ Scale*, G. Kiczales, D. Russell, and B. Woolf, Eds. 195–204.

POLITI, A. 2013. Lyapunov exponent. *Scholarpedia 8,* 3, 2722.

PRICE, T., ZHI, R., AND BARNES, T. 2017. Evaluation of a data-driven feedback algorithm for open-ended programming. In *Proceedings of the 10th International Conference on Educational Data Mining*, X. Hu, T. Barnes, and P. Inventado, Eds. 192–197.

PRICE, T. W., DONG, Y., ZHI, R., PAASSEN, B., LYTLE, N., CATETÉ, V., AND BARNES, T. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education 29,* 3, 368–395.

QIAN, Y. AND LEHMAN, J. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education 18,* 1, 1–25.

REDDY, S., LABUTOV, I., AND JOACHIMS, T. 2016. Latent skill embedding for personalized lesson sequence recommendation. *arXiv 1602.07029*.

RIVERS, K. AND KOEDINGER, K. R. 2012. A canonicalizing model for building programming tutors. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems, (ITS 2012)*, S. A. Cerri, W. J. Clancey, G. Papadourakis, and K. Panourgia, Eds. 591–593.

RIVERS, K. AND KOEDINGER, K. R. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education 27,* 1, 37–64.

ROBINS, A., ROUNTREE, J., AND ROUNTREE, N. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education 13,* 2, 137–172.

SCARSELLI, F., GORI, M., TSOI, A. C., HAGENBUCHNER, M., AND MONFARDINI, G. 2009. The graph neural network model. *IEEE Transactions on Neural Networks 20,* 1, 61–80.

TAI, K. S., SOCHER, R., AND MANNING, C. D. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistic*, Y. Matsumoto, C. Zong, and M. Strube, Eds. 1556–1566.

TRUONG, N., ROE, P., AND BANCROFT, P. 2004. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education*, R. Lister and A. Young, Eds. 317–325.

WILES, R., DURRANT, G., BROE, S. D., AND POWELL, J. 2009. Methodological approaches at PhD and skills sought for research posts in academia: A mismatch? *International Journal of Social Research Methodology 12,* 3, 257–269.

ZHANG, K. AND SHASHA, D. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing 18,* 6, 1245–1262.

ZHI, R., PRICE, T. W., LYTLE, N., DONG, Y., AND BARNES, T. 2018. Reducing the state space of programming problems through data-driven feature detection. In *Proceedings of the second Educational Data Mining in Computer Science Education Workshop*, D. Azcona, S. Hsiao, N.-T. Le, J. Stamper, and M. Yudelson, Eds.

ZIMMERMAN, K. AND RUPAKHETI, C. R. 2015. An automated framework for recommending program elements to novices (N). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, M. Cohen, L. Grunske, and M. Whalen, Eds. 283–288.

APPENDIX

A. TRAINING SCHEME FOR AST2VEC

To train ast2vec, we minimized the variational autoencoding (VAE) loss of Kingma and Welling (2013). Let $\hat{x}_1, \ldots, \hat{x}_N$ be a training dataset of syntax trees. Then, the VAE loss expresses two training goals: First, we want to maximize the probability $p_\psi\big(\hat{x}_i \big| \phi(\hat{x}_i)\big)$ that every tree $\hat{x}_i$ gets decoded back to itself after being (noisily) encoded as $\phi(\hat{x}_i)$. Second, we want to make sure that the distribution of the noisy code $\phi(\hat{x}_i)$ remains close to the standard normal distribution.

More precisely, the VAE loss has the following form:

$$\ell(\phi, \psi) = \sum_{i=1}^{N} -\log\Big[p_\psi\big(\hat{x}_i \big| \phi(\hat{x}_i)\big)\Big] + \beta \cdot D_{KL}\Big(\phi(\hat{x}_i)\Big), \tag{9}$$

where ϕ is the noisy version of the encoder we want to train, ψ is the decoder we want to train, β is a hyper-parameter to regulate how smooth we want our coding space to be, and $D_{KL}\Big(\phi(\hat{x}_i)\Big)$ is the Kullback-Leibler divergence between the distribution of the noisy neural code $\phi(\hat{x}_i)$ and the standard normal distribution - i.e., it punishes if the code distribution becomes too different from a standard normal distribution (Kingma and Welling, 2013).

For this training procedure to work we need *probabilistic* versions of both our encoder ϕ and our decoder ψ. For the encoder, we achieve this goal by producing two n-dimensional outputs, $\mu(\hat{x}_i)$ and $\sigma(\hat{x}_i)$ and setting the code $\phi(\hat{x}_i) = \mu(\hat{x}_i) + \epsilon \odot \sigma(\hat{x}_i)$, where ϵ is an n-dimensional vector of standard normal noise and $\odot$ is element-wise multiplication. After training, we can throw away the auxiliary output $\sigma(\hat{x}_i)$ and set $\phi(\hat{x}_i) = \mu(\hat{x}_i)$. With this setting, the Kullback-Leibler divergence becomes

$$D_{KL}\Big(\phi(\hat{x}_i)\Big) = \sum_{j=1}^{n} \mu(\hat{x}_i)_j^2 + \sigma(\hat{x}_i)_j^2 - \log\big[\sigma(\hat{x}_i)_j^2\big].$$

For the decoder, we need a distribution $p_\psi(\hat{x}|\vec{x})$ that determines the probability of vector $\vec{x}$ getting decoded to tree $\hat{x}$. We construct this probability distribution as follows. First, we define a probability $p_A(x|\vec{x})$ of syntactic element x getting decoded from vector $\vec{x}$ by grammar symbol A. We set this to the softmax distribution:

$$p_A(x|\vec{x}) = \frac{\exp\big[h_A(x|\vec{x})\big]}{\sum_y \exp\big[h_A(y|\vec{x})\big]}.$$

In other words, we deem it as more likely that x is selected if the score $h_A(x|\vec{x})$ is much higher than for any other possible syntactic element y. Then, we define the overall probability $p_\psi(\hat{x}|\vec{x})$ as the product of $p_A(x|\vec{x})$ over all decoding decisions that would be needed to decode tree $\hat{x}$ from vector $\vec{x}$, following Equation 3. The negative logarithm of this probability then coincides with the cross-entropy loss between the 'correct' decoding decisions to achieve tree $\hat{x}$ and the decoding decisions that our current model would choose according to p_A.

To minimize the VAE loss in Equation 9, we apply an Adam optimizer (Kingma and Ba, 2015). In more detail, we perform a sequence of training epochs. In each of these epochs, we randomly sample a mini-batch of $N = 32$ abstract syntax trees $\hat{x}_1, \ldots, \hat{x}_N$, encode them as vectors $\phi(\hat{x}_i)$ using the current encoder ϕ, add random Gaussian noise $\epsilon(\hat{x}_i)$, and then compute the

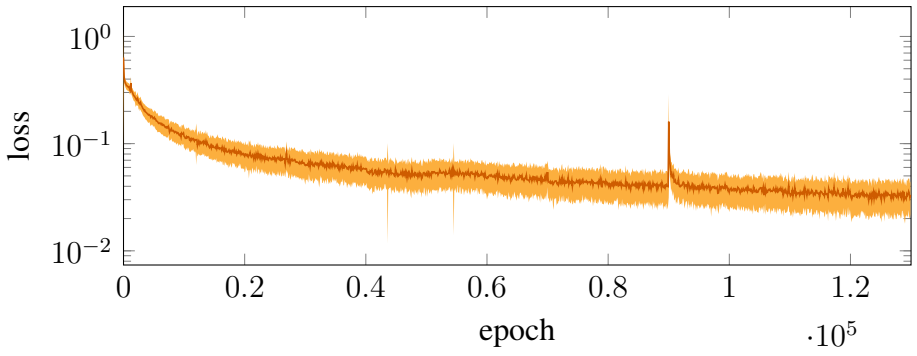

Figure 13: The learning curve for our recursive tree grammar autoencoder trained on $448,992$ Python programs recorded in the 2018 NCSS beginners challenge of Grok Learning. The dark orange curve shows the loss of Equation 9 over the number of training epochs. For better readability, we show the mean over 100 epochs, enveloped by the standard deviation (light orange). Note that the plot is in log scaling, indicating little change toward the end.

probabilities $p_\psi\big(\hat{x}_i \big| \phi(\hat{x}_i) + \vec{\epsilon}_i\big)$ as described above. With these probabilities, we compute the loss in Equation 9. Note that this loss is differentiable with respect to all our neural network parameters. Accordingly, we can use the automatic differentiation mechanisms of PyTorch (Paszke et al., 2019) to compute the gradients of the loss with respect to all parameters and then perform an Adam update step (Kingma and Ba, 2015). We set the dimensionality n to 256, the smoothness parameter β to 10^{-3}, and the learning rate to 10^{-3}.

We trained ast2vec on $448,992$ Python programs recorded as part of the the National Computer Science School (NCSS)[8], an Australian educational outreach program. The course was delivered by the Grok Learning platform[9]. Each offering was available to mostly Australian school children in Years 5–10, with curriculum-aligned educational slides and sets of exercises released each week for five weeks. For training ast2vec, we used all programs that students tried to run in the 2018 'beginners' challenge. After compilation, we were left with $86,991$ unique abstract syntax trees. We performed training for $130,000$ epochs (each epoch with a mini-batch of 32 trees), which corresponds to roughly ten epochs per program ($32 \cdot 130,000 = 4,160,000$). By inspecting the learning curve of the neural net, we notice that the loss has almost converged (refer to Figure 13). All training was performed on a consumer-grade laptop with a 2017 Intel core i7 CPU[10] and took roughly one week of real time.

B. Proofs for the dynamical system analysis

Theorem 1. *Let $f(\vec{x}) = \vec{x} + \boldsymbol{W} \cdot (\vec{x}_* - \vec{x})$ for some matrix $\boldsymbol{W} \in \mathbb{R}^{n \times n}$. If $|1 - \lambda_j| < 1$ for all eigenvalues λ_j of $\boldsymbol{W}$, f asymptotically converges to $\vec{x}_*$.*

Proof. We note that the theorem follows from general results in stability analysis, especially Lyapunov exponents (Politi, 2013). Still, we provide a full proof here to give interested readers

[8] https://ncss.edu.au

[9] https://groklearning.com/challenge/

[10] In contrast to other neural networks, recursive neural nets can not be trained on GPUs because the computational graph is unique for each tree.

insight into how such an analysis can be done.

In particular, let $\vec{x}_1$ be any n-dimensional real vector. We know wish to show that plugging this vector into f repeatedly yields a sequences $\vec{x}_1, \vec{x}_2, \ldots$ with $\vec{x}_{t+1} = f(\vec{x}_t)$ which asymptotically converges to $\vec{x}_*$ in the sense that

$$\lim_{t\to\infty} \quad \vec{x}_t = \vec{x}_*.$$

To show this, we first define the alternative vector $\hat{x}_t := \vec{x}_t - \vec{x}_*$. For this vector we obtain:

$$\hat{x}_{t+1} = \vec{x}_{t+1} - \vec{x}_* = f(\vec{x}_t) - \vec{x}_* = \vec{x}_t + \boldsymbol{W} \cdot (\vec{x}_* - \vec{x}_t) - \vec{x}_* = \big(\boldsymbol{I} - \boldsymbol{W}\big) \cdot \hat{x}_t = \big(\boldsymbol{I} - \boldsymbol{W}\big)^t \cdot \hat{x}_1.$$

Further, note that our desired convergence of $\vec{x}_t$ to $\vec{x}_*$ is equivalent to stating that $\hat{x}_t$ converges to zero. To show that $\hat{x}_t$ converges to zero, it is sufficient to prove that the matrix $\big(\boldsymbol{I} - \boldsymbol{W}\big)^t$ converges to zero.

To show this, we need to consider the eigenvalues of our matrix $\boldsymbol{W}$. In particular, let $\boldsymbol{V} \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{V}^{-1} = \boldsymbol{W}$ be the eigenvalue decomposition of $\boldsymbol{W}$, where $\boldsymbol{V}$ is the matrix of eigenvectors and $\boldsymbol{\Lambda}$ is the diagonal matrix of eigenvalues $\lambda_1, \ldots, \lambda_n$. Then it holds:

$$\begin{aligned}\big(\boldsymbol{I} - \boldsymbol{W}\big)^t &= \big(\boldsymbol{V} \cdot \boldsymbol{V}^{-1} - \boldsymbol{V} \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{V}^{-1}\big)^t = \big(\boldsymbol{V} \cdot (\boldsymbol{I} - \boldsymbol{\Lambda}) \cdot \boldsymbol{V}^{-1}\big)^t \\ &= \boldsymbol{V} \cdot (\boldsymbol{I} - \boldsymbol{\Lambda}) \cdot \boldsymbol{V}^{-1} \cdot \boldsymbol{V} \cdot \ldots \cdot \boldsymbol{V}^{-1} \cdot \boldsymbol{V} \cdot (\boldsymbol{I} - \boldsymbol{\Lambda}) \cdot \boldsymbol{V}^{-1} = \boldsymbol{V} \cdot (\boldsymbol{I} - \boldsymbol{\Lambda})^t \cdot \boldsymbol{V}^{-1}.\end{aligned}$$

Now, let us consider the matrix power $(\boldsymbol{I} - \boldsymbol{\Lambda})^t$ in more detail. Because this is a diagonal matrix, the power is just applied elementwise. In particular, for the jth diagonal element we obtain the power $(1 - \lambda_j)^t$. In general, the jth eigenvalue can be complex-valued. However, the absolute value still behaves like $|(1 - \lambda_j)^t| = |1 - \lambda_j|^t$. Now, since we required that $|1 - \lambda_j| < 1$ we obtain

$$\lim_{t\to\infty} \quad |1 - \lambda_j|^t = 0 \qquad \Rightarrow \qquad \lim_{t\to\infty} \quad \boldsymbol{V} \cdot (\boldsymbol{I} - \boldsymbol{\Lambda})^t \cdot \boldsymbol{V}^{-1} = \boldsymbol{0},$$

which implies our desired result. □

Theorem 2. *Problem 7 has the unique solution in Equation 8.*

Proof. We solve the problem by setting the gradient to zero and considering the Hessian.

First, we compute the gradient of the loss with respect to $\boldsymbol{W}$:

$$\begin{aligned}&\nabla_{\boldsymbol{W}} \sum_{t=1}^{T-1} \|\vec{x}_t + \boldsymbol{W} \cdot (\vec{x}_* - \vec{x}_t) - \vec{x}_{t+1}\|^2 + \lambda \cdot \|\boldsymbol{W}\|_{\mathcal{F}}^2 \\ =&2 \cdot \sum_{t=1}^{T-1} \big(\vec{x}_t + \boldsymbol{W} \cdot (\vec{x}_* - \vec{x}_t) - \vec{x}_{t+1}\big) \cdot (\vec{x}_* - \vec{x}_t)^T + 2 \cdot \lambda \cdot \boldsymbol{W}\end{aligned}$$

Setting this to zero yields

$$\begin{aligned}\boldsymbol{W} \cdot \Big(\sum_{t=1}^{T-1} (\vec{x}_* - \vec{x}_t) \cdot (\vec{x}_* - \vec{x}_t)^T + \lambda \cdot \boldsymbol{I}\Big) &= \sum_{t=1}^{T-1} (\vec{x}_t - \vec{x}_{t+1}) \cdot (\vec{x}_* - \vec{x}_t)^T \\ \iff \quad \boldsymbol{W} \cdot \big(\boldsymbol{X}_t^T \cdot \boldsymbol{X}_t + \lambda \cdot \boldsymbol{I}\big) &= \big(\boldsymbol{X}_{t+1} - \boldsymbol{X}_t\big)^T \cdot \boldsymbol{X}_t\end{aligned}$$

For $\lambda > 0$, the matrix $\boldsymbol{X}_t^T \cdot \boldsymbol{X}_t + \lambda \cdot \boldsymbol{I}$ is quaranteed to be invertible, which yields our desired solution.

It remains to show that the Hessian for this problem is positive definite. Re-inspecting the gradient, we observe that the matrix $\boldsymbol{W}$ occurs only as a product with the term $\sum_{t=1}^{T-1}(\vec{x}_* - \vec{x}_t) \cdot (\vec{x}_* - \vec{x}_t)^T$, which is positive semi-definite, and the term $\lambda \cdot \boldsymbol{I}$, which is strictly positive definite. Hence, our problem is convex, and our solution is the unique global optimum. □