Chapter 1

Bayesian Clustering of Student Solutions

The original vision of OverCode was to discover pedagogically valuable themes of variation within thousands of student solutions to the same programming problem. ~~Each~~ It summarizes the most common solutions well because they are grouped into large clusters. It also produces a long tail of small clusters. Pulling out themes of variation from the long tail can be challenging.

There are a variety of clustering methods that could be applied to pulling themes out of the long tail of small OverCode clusters. Some methods create hierarchical clusterings. At higher levels of the hierarchy, clusters contain a larger diversity of solutions. Some methods break solutions down into clusterable components or allow solutions to partially belong to several clusters, so that each solution does not need to be strictly associated with a single cluster. Bayesian methods treat student solutions as examples sampled from some underlying distribution of student solutions. Each of these families of methods have appealing properties.

The recent development of Bayesian clustering methods built explicitly for interpretability spurred my exploration of Bayesian methods for clustering student solutions. This chapter explores the application of two Bayesian clustering methods to the normalized solutions produced by OverCode.

1.1 OverCode as Clustering Algorithm

In OverCode, each Python solution is normalized ~~before clustering and represented as a sequence of normalized lines of code~~and clustered. All solutions that share the same set of normalized lines of code are clustered together. The clustering is a strict partition of the solutions. This clustering process is hard and agglomerative but not hierarchical. It was designed with interpretability in mind. Hierarchical clustering may also be appropriate in this context, as long as human interpretability does not suffer. The OverCode clustering pipeline is also interactive, because the results are shown in an interface that allows the human to tell the system that some lines of code are not different in meaningful ways. If any clusters differ only by those lines of code that the human considers to be the same, those clusters will be merged.

In OverCode, what can vary and what is invariant across all the solutions within a stack is clear, just by reading the platonic solution that represents the stack. Syntactic differences split apart groups of solutions into smaller stacks so that each stack's platonic solution matches the syntax used in the solutions in represents. This can also produce more stacks than teachers can read. OverCode does normalize and cluster correct solutions so that they can be more easily understood as a group, but it does not pull out larger themes of variation as clearly as initially hoped.

~~However, OverCode's normalized stacks do exhibit statistical regularity.~~

1.2 The Applicability of Bayesian Clustering Methods

Given that variation theory is interested in dimensions of variation and consistency that characterize all possible instantiations of an idea, statistical methods warrant further investigation. Latent variable models are a type of statistical model that attempt to explain variation in a dataset based on underlying factors. With the right choice of features and model, a latent variable model may be able to capture underlying design choices. ~~In this~~

~~section, two~~ Two latent variable models have been explored in a preliminary way for this purpose: the Bayesian Case Model (BCM) [**?**] and Latent Dirichlet Allocation (LDA) [**?**].

BCM was selected because, like the original OverCode pipeline, it clusters solutions and produces a single solution to represent the entire cluster. Like OverCode, it also indicates the features that characterize each cluster. While OverCode displays the platonic solution that shares the same set of normalized lines with all other members of the cluster, BCM learns a subspace of features that it determines are most characteristic of the solutions within that cluster.

BCM has an interactive variant called iBCM. iBCM allows the teacher to directly modify the prototype and the subspace chosen by BCM if it disagrees with their domain knowledge or preferences. This teacher action triggers a rerun of BCM with the modifications taken into account.

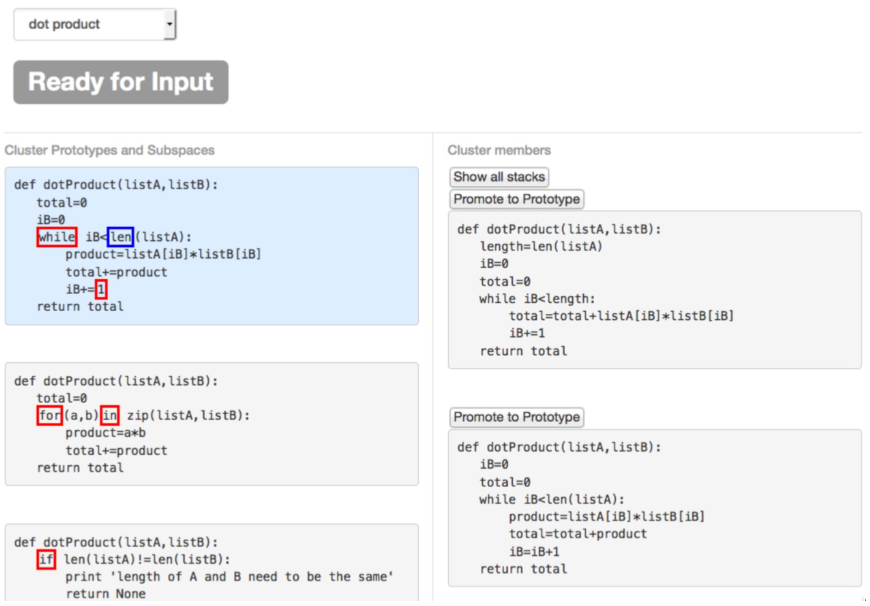
Internally, BCM depends on a mixture model with a Dirichlet prior. Rather than find a cluster for each entire solution, mixture models can learn clusters of features that co-exist across some subset of all solutions. The concentration parameter of the BCM Dirichlet prior is set to promote sparsity~~, i. e.,~~ . That means that the mixture distributions over solutions ~~that~~ will be biased to have the majority of their probability mass on a single mixture component. BCM then assigns each solution to a cluster according to its most probable mixture component.

Since teachers can be poor at clustering solutions, i.e., inconsistent with each other [**?**], it may also be difficult for any algorithm to find a clustering that appeals to most teachers. However, there are a statistical models, lie LDA, that cluster components of solutions instead of entire solutions. LDA was selected as an alternative to BCM in order to explore how solution component clustering might better support teachers who do not agree on whole-solution clusterings.

BCM was built explicitly with interpretability in mind, but it is still not clear whether LDA or BCM are more interpretable in the context of clustering OverCode stacks. LDA has some known interpretability problems. As part of its training process, LDA learns distributions over features, also known as *mixture components*. It can be difficult to interpret exactly what an LDA mixture component means just by looking at its distribution over features.

1.3 Clustering Solutions with BCM

BCM was applied to the platonic solutions generated by OverCode. The platonic solutions encode both static and dynamic information, i.e., the syntax carries the static information and the variable naming encodes dynamic information. The platonic solution is tokenized and represented as a binary vector indicating the existence of the features, including renamed variables and language-specific keywords, such as normalized variable names like `listA` and Python keywords like `assert` and `while`. The result is a clustering of OverCode platonic solutions.



```
def dotProduct(listA,listB):
    total=0
    iB=0
    while iB<len(listA):
        product=listA[iB]*listB[iB]
        total+=product
        iB+=1
    return total
```

```
def dotProduct(listA,listB):
    total=0
    for(a,b)in zip(listA,listB):
        product=a*b
        total+=product
    return total
```

```
def dotProduct(listA,listB):
    if len(listA)!=len(listB):
        print 'length of A and B need to be the same'
        return None
```

```
def dotProduct(listA,listB):
    length=len(listA)
    iB=0
    total=0
    while iB<length:
        total=total+listA[iB]*listB[iB]
        iB+=1
    return total
```

```
def dotProduct(listA,listB):
    iB=0
    total=0
    while iB<len(listA):
        product=listA[iB]*listB[iB]
        total=total+product
        iB=iB+1
    return total
```

Figure 1-1: This interface displays iBCM clusterings of OverCode platonic solutions and affords user feedback on cluster quality. The solutions on the left are cluster prototypes. The blue solution is the selected cluster whose members are shown in a scrollable list on the right-hand side. The tokens contained in red boxes are features that BCM identifies as characteristic of the cluster represented by that prototype. When a user hovers ~~their~~ the cursor over a keyword or variable name, e.g., `len`, it is highlighted in a blue rectangle, indicating that it can be clicked.

1.3.1 Preliminary Evaluation

BCM was run on three different programming problems selected from those previously analyzed in Chapter **??**. An interface, shown in Figure **??**, displayed the results, with iBCM continuing to run in the backend, ready to update the clustering in response to teacher feedback. The interface afforded promoting a member of a cluster to be its prototype and clicking on a token within a prototype, such as a variable name or keyword, to toggle whether or not it is labeled a characteristic feature of that cluster.

In order to determine whether or not these clusterings and interface affordances are useful to teachers, a small pilot study was run on three teachers of introductory Python programming. For the pilot, two more interfaces were introduced as controls. One is a duplicate of the control interface used in the OverCode studies, i.e., a long list of syntax highlighted raw solutions in a random order in the browser. The second interface differs from the previous control by one factor: the raw solutions are replaced with their platonic equivalents.

The teachers were each given sets of Python solutions to view in each of the three interfaces. For each problem, they were asked to create a grading rubric and provide helpful comments for the students of the type that might be mentioned in a class-wide forum post.

The teachers appreciated the fact that BCM gave some structure to the space of solutions. Rather than a long list of solutions, the interface suggested distinct subpopulations of solutions within the list. However, subjects did not fully understand the probabilistic nature of the clustering method. The presence of a single intruder, i.e., a solution that the teacher believed did not belong in a cluster, caused confusion. This could be ameliorated by giving teachers more ways to modify the clustering, e.g., allowing teachers the option to kick an intruder out of a cluster and rerun BCM. Subjects also requested richer or higher-level features than variable names and keywords. Been Kim's PhD thesis [**?**] describes a follow-up full user study comparing the efficacy of BCM and iBCM on clustering OverCode platonic solutions.

1.4 Clustering Solution Components with LDA

Other researchers have documented a lack of agreement across human-made clusters of student solutions [**?**]. One possible explanation for poor consistency is that solutions are mixtures of design choices and teachers care differently about various aspects of solutions. If student A writes a solution with a well-written loop and extraneous statements while student B writes a solution with extra loops but otherwise very clean code, teachers can reasonably disagree about which cluster each whole solution should be placed in, depending on whether they believe inefficient control flow or extraneous statements are worse.

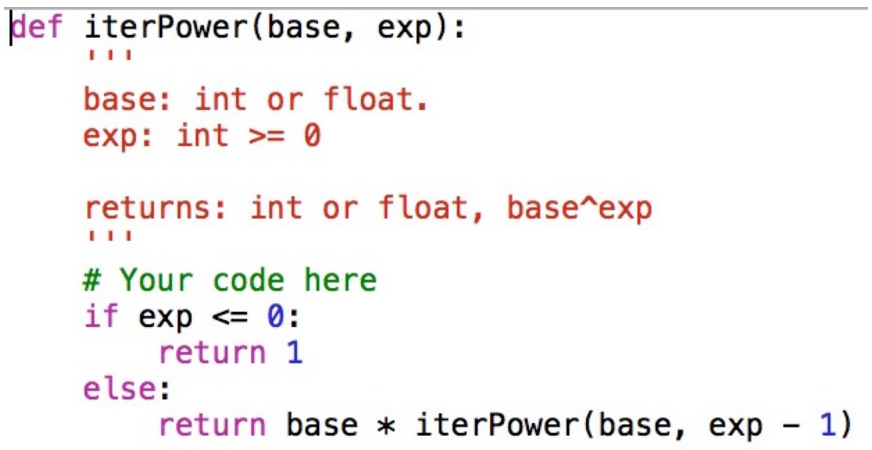
Instead of trying to approximate clusterings that humans do not even agree on, it may be more useful to model solutions as mixtures of good and bad design choices. While more sophisticated mixture models' assumptions may ultimately be more appropriate, LDA [**?**] as implemented in the Gensim toolbox [**?**] was chosen as the model to evaluate in this preliminary work.

Like BCM, LDA was run on the platonic solutions. However, the representation of these solutions was also changed: in order to pull out higher-level patterns in approach, rather than lower-level patterns in syntax, solutions were represented solely by the behavior of the variables within them.

As described in the chapter on OverCode, the OverCode analysis pipeline executes all programs on a common set of test cases and records the sequence of values taken on by each variable in the program. OverCode assumes that variables in different programs that transition through the same sequence of values on the same test cases are in fact fulfilling the same semantic role in the program.

Figure **??** shows the sequences of variable values recorded by OverCode while executing `iterPower(5,3)` as defined in Figures **??**, **??**, and **??**:

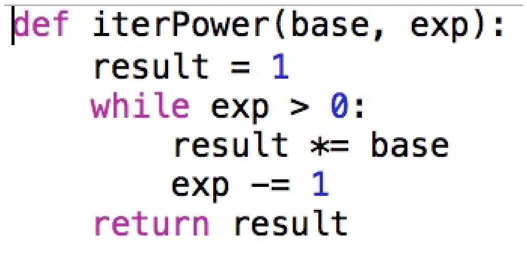
In these examples, the input argument `base` would be considered a variable common to all three programs, but the input argument `exp` would not be. The variable `result` would



```
def iterPower(base, exp):
    '''
    base: int or float.
    exp: int >= 0

    returns: int or float, base^exp
    '''
    # Your code here
    if exp <= 0:
        return 1
    else:
        return base * iterPower(base, exp - 1)
```

Figure 1-2: Example of a recursive student solution.



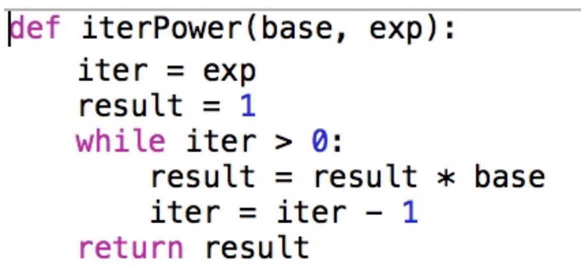
```
def iterPower(base, exp):
    result = 1
    while exp > 0:
        result *= base
        exp -= 1
    return result
```

Figure 1-3: Example of a student solution using the Python keyword `while`.

also be considered a common variable shared across just the definitions in Figure **??** and Figure **??**. This allows us to distinguish between programs that calculate the answer in semantically distinct ways, without discriminating between the low-level design decisions about syntax.

LDA is often applied to corpora of textual documents, where the corpus is represented as a $W \times \mathrm{N}$ term-by-document matrix of counts, where W is the vocabulary size across all documents and N is the number of documents. In this representation, the document is represented a bag of word counts, i.e., how many times each word appears in the document. Following this analogy, solutions are represented as a bag of variable behaviors. The matrix representing the solutions in Figures **??** through **??** executed on `iterPower(5,3)` is shown in Table **??**.

Note that, while the entries in Table **??** only take on values 0 or 1, more complicated definitions may have n instances of, e.g., a variable that takes on the sequence of values `3,2,1,0`.



```
def iterPower(base, exp):
    iter = exp
    result = 1
    while iter > 0:
        result = result * base
        iter = iter - 1
    return result
```

Figure 1-4: Example of a student solution using the Python keyword `while`, where the student has not modified any input arguments, i.e., better programming style.

- **Figure ??**
  - `exp:  3, 2, 1, 0, 1, 2, 3`
  - `base:  5`
- **Figure ??**
  - `exp:  3, 2, 1, 0`
  - `base:  5`
  - `result:  1, 5, 25, 125`
- **Figure ??**
  - `exp:  3`
  - `base:  5`
  - `iter:  3, 2, 1, 0`
  - `result:  1, 5, 25, 125`

Figure 1-5: The sequences of variable values recorded by OverCode while executing `iterPower(5,3)`.

In that case, there could be an n in the `3,2,1,0` column for the row corresponding to that solution, or it can be left as a binary indicator.

In order to run LDA, student solutions to a particular problem are first run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline produces a set of platonic solutions and a set of features for each solution, including which variable sequences were observed during execution. A script extracts the variable-by-solution matrix. LDA is then run repeatedly with different values for the parameter that sets the number of latent mixture components and then inspected by hand. The results of running LDA on platonic solutions is inspected by hand because cluster validity metrics like perplexity and held-out likelihoods are not necessarily good proxies for human interpretability [**?**].

Inspection by hand can also be challenging. When LDA learns mixture components in this

Table 1.1: Variable-by-Solution Matrix for Programs, where variables are uniquely identified by their sequence of values while run on a set of test case(s)

SOLUTION	5	`1,5,... 25,125`	`3,2,1,0, ...1,2,3`	`3,2,1,0`	3
FIGURE **??**	1	0	1	0	0
FIGURE **??**	1	1	0	1	0
FIGURE **??**	1	1	0	1	1

context, those components are distributions over variable behaviors. Rather than inspect these distributions, entire solutions containing high amounts of that mixture component were inspected instead. By comparing entire solutions with high and low amounts of each component, one can infer what each learned component captured. This method for understanding the results is similar to the recommendations of variation theory.

1.4.1 Preliminary Evaluation

3875 student solutions to `iterPower` were run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline produced a set of 977 platonic solutions and sets of features for each solution. LDA was run on the resulting variable-by-solution matrix, using different parameter values.

The LDA results were inspected by hand. One topic comparison captured the difference between solutions like Figure **??** and **??**. Another topic comparison exposed the difference between the subpopulation of solutions with unnecessary conditional statements and the common, more concise solution.

1.5 Discussion and Future Work

LDA applied to a variable-by-solution matrix is a promising method for identifying variation within corpora of solutions to the same programming problem. However, the assumptions made in LDA, such as the independence of mixture components, and requirements, such as explicitly setting the number of mixture components beforehand, may mean that other mixture models, such as the Correlated Topic Model [**?**] or the Hierarchical Dirichlet Process [**?**] will ultimately be a better model fit for this purpose. In the future, new user interfaces would be helpful for this task, especially one which helps humans compare the output of models with different parameter values.

1.6 Conclusion

The clustering described in the original OverCode work was relatively limited in scope, but it did produce platonic solutions that can be used for more statistically sophisticated clustering techniques and as a starting point for helping graders understand and grade incorrect student solutions by hand.