Class Introspection: A Novel Technique for Detecting Unlabeled Subclasses by Leveraging Classifier Explainability Methods

Patrick Kage

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2021

Abstract

A common problem for data science is the detection of unlabeled latent subclasses within a dataset. For example, a classifier predicting labels for images of apples and oranges may treat red and green apples differently—which may be undesired behavior if it is not intentional. This subclass discovery is difficult to perform with state-of-the-art techniques: either they are limited to detecting very small numbers of outliers or they lack the statistical power to deal with complex data. This paper proposes a solution to this subclass discovery problem: by leveraging instance explanation methods, an existing classifier can be extended to detect latent classes via differences in the classifier's internal decisions about each instance. This works not only with simple classification techniques but also with deep neural networks, allowing for a powerful and flexible approach to detecting latent structure within datasets. This paper also contains a pipeline for analyzing classifiers automatically, and a web application for interactively exploring the results from this technique.

Acknowledgements

Thanks to my advisor, Dr. Pavlos Andreadis, for his unending support and guidance throughout this project.

Additional thanks to Jake Lee from NASA's Jet Propulsion Laboratory, for helping figure out computer vision approaches to saliency map interpretation.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Training classifiers for machine learning tasks requires that the data is accurately and completely labeled for a specific application. However, in the real world there is often more structure to the data than is labeled—and this can have real-world consequences to how the model performs in production processes. Within each labeled class, there can be significant variations that the model picks up on but is invisible to the user—this is *latent structure* within the class. For an example of this latent structure, consider a hypothetical classifier to determine whether an image contains apples or oranges. Oranges tend to be uniformly orange, but apples can come in more than one color: red, green, yellow, etc. If the labels for the dataset are just `apple` and `orange`, then the information about the color of the apples is lost. The intuition here is simple: the classifier may know about the color difference between two types of apples, but still label both apples due to the training data available to it. Therefore, by using explainability techniques, a human can detect the different unlabeled subclasses by analyzing *how* the classifier determines the class of an instance.

As a less trivial but more impactful example, clinical trial results interpreting the efficacy of a drug's treatment via a classifier may not fully capture all the subgroups in the input. Are the reasons Person A and Person B respond to a specific round of treatment similar? How about why Person C did not respond? This may be down to some specific structure in the input data which may not be fully captured by training data labeling.

With current methods, it is difficult to determine this latent structure—especially with complex data. Current state-of-the-art methods generally require either that an entirely new classifier is trained (as in the case of mixture models[24]), that instances be categorized manually (as in subgroup analysis[12]), or they are only suited to discovering individual anomalous instances (as in commonality metrics[18]). These models are not one-size-fits-all, and require a separate (new) model to detect latent structure.

1.2 Objectives

This project aims to provide a solution for the latent structure problem by leveraging explainability techniques to detect latent (unlabeled) subclasses in the input data, using a novel approach dubbed **class introspection**. This technique compares a classifiers decision making process for each point in a dataset, and within each predicted class performs clustering over the local model explanations. Crucially, this *re-uses the existing classifier*, and does not require a separate model for detecting latent structure. At a high level, the decision-making process of a classifier model will be different for different inputs—leading to clusters corresponding to fragmentation within input classes. This provides a level of auditability on the model training process, both by ensuring that models are producing results for the correct reasons and by allowing for the detection of deficiencies in the model setup priors; by detecting latent structure, possible errors in labeling can be detected in model training. Additionally, this technique is agnostic to both the architecture of the particular classifier model used and the particular local explanation method, allowing for use in even black-box environments (where no knowledge of the classifier's internals is required).

The objectives of the project are as follows:

1. Implement a pipeline to detect latent structure in datasets using class introspection (using artificially-created latent structure).
2. Demonstrate a class introspection pipeline in several different datasets and several different models.
3. Compare class introspection performance to a baseline method.
4. Create a toolchain for detecting latent structure using class introspection methods.

This paper will discuss the background necessary for class introspection, the methodology itself, and the series of experiments over which the methodology was refined. Additionally, a discussion of results and limitations of the method are included with an overview of future work in this area.

Chapter 2

Background

2.1 Overview

In this section, we will discuss the algorithms and methods that are fundamental to class introspection (namely, explainability and clustering methods). Additionally, this section discusses several existing algorithms for detecting structure in data and their limitations as compared to class introspection.

2.2 Explainability

A key issue facing machine learning is explainability (XAI). In current state-of-the-art machine learning models, the model outputs are generated opaquely—that is, the reasons that the model chooses to assign one label rather than another are inscrutable from the outside. While this may be fine in trivial applications such as face detection, in safety-critical applications the reasoning for why a model produces the outputs it does can be a literal case of life and death. Explainability allows for a model to be audited, and for faulty behaviors to be explained and calibrated away[21]. Even more crucially, explainability allows for a model's decisions to be trusted by other agents (like humans)[16].

2.2.1 What is an explanation?

Explanations are, at their core, simply an indication of which input features are relevant for a specific output from a network, and for each how strongly (or not) these features contributed to the output. Typically this is represented numerically, where each input feature is weighted by its importance to the overall classification with respect to the other features[16].

In tabular data, numerically weighting input features trivially shows which inputs are important as the feature is clearly defined (as seen in Figure 2.1). Image data is more complex, as the features do not carry the same amount of information per-feature as tabular data; explanations for this type of data show which regions of the image were important to a model's classification (see Figure 2.2).

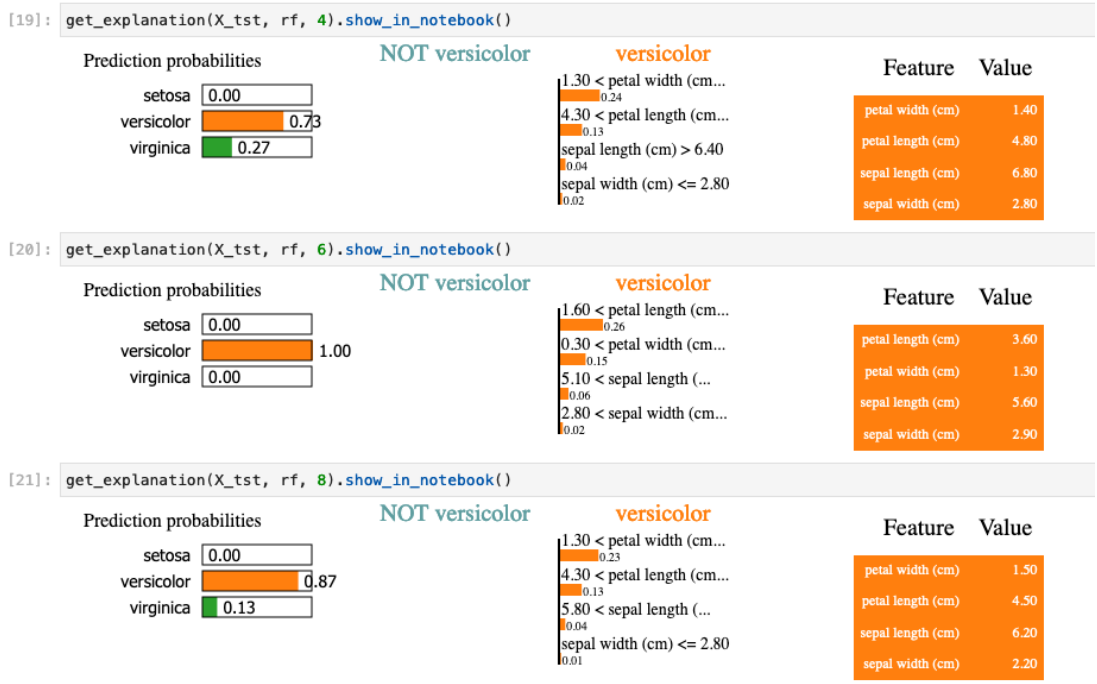

Figure 2.1: Several examples of explanation of tabular data from the Iris dataset (see Section 4.2), created with LIME. Note how due to there being no evidence of these instances not being in the `versicolor` class (i.e. all features point towards the `versicolor` class), the `NOT versicolor` column is empty.

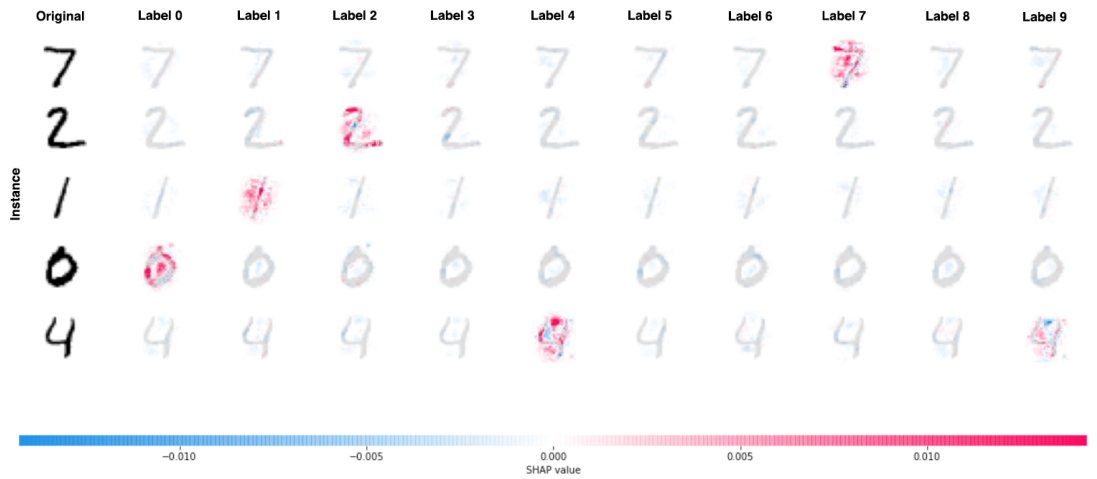

Figure 2.2: Example of a series of explanations of image data. Here, regions of the image that contribute positively towards the classification are red, and similarly blue regions denote negative contributions. Note that SHAP creates explanations for each instance for each label, regardless of true label. For more information, see Section 2.2.4.

2.2.2 Inherently Interpretable Models

The easiest way to generate explanations for models is by using models that are inherently interpretable. There are two common model architectures with this feature:

regression models and decision trees. Decision tree outputs can be simply explained by following the chain of decisions from the root node down to the eventual leaf representing the classification. Regression models are similarly simple, with linear regression and logistic regression:

$$\text{linreg}(\mathbf{x};\mathbf{w},b) = \mathbf{w}^\top \mathbf{x} + b \qquad \text{logreg}(\mathbf{x};\mathbf{w},b) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

We can see that for each case, there is a one-to-one correspondence between the weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$. Therefore, the relative importance of each input feature in $\mathbf{x}$ is directly encoded in the weights of the model—exactly an explanation!

Of course, using regression models or decision trees may not be an optimal strategy as these models are very limited in their capabilities. Another option is to converting an existing model as a whole into a more easily interpretable model architecture. For example, an interpretable model can be created with *Self-Explanatory Neural Networks* (SENNs) which are an extension of logistic regression[20]. This can use very few weights, and it is easy to determine the reasoning for any given classification as the weights correspond to a positive or negative linear combination of inputs. This approach, however, is limited in that some algorithms are not well-suited to interpretability. Algorithms such as deep neural networks are uninterpretable, and their performance cannot easily be matched by interpretable models.

2.2.3 Local Outcome Modeling

If the model to be explained cannot be converted to an easily-interpretable model, it is still possible to create an explanation of its behavior using black-box explanation methods.

The simplest black-box approach is the Leave-One-Out (LOO) algorithm[23]. LOO simply segments the input features, and for each segment zeros out the segment and runs the model inference to determine how much the output changes. Despite LOO's apparent simplicity, this strategy is surprisingly effective at determining the salient regions of an input, and being a black-box algorithm means it can be broadly applicable to a diverse variety of algorithms[23]. Careful selection of segmentation algorithms can lead to very accurate saliency maps (see Figure 2.3). However, this flexibility comes at a cost. LOO is expensive to compute, as each segment requires a re-inference; additionally, this method does not take into account inter-dependencies between regions[23]. Additionally, this does not give per-feature saliency mapping, rather operating over superpixels—useful for individual explanations, but less so for comparing explanations between instances.

A more robust approach is to train a smaller model to explain individual predictions of a dataset[17]. This local approach has the benefit that the overarching model can be a black box while still providing explanations of the model behavior. This works because even if the decision surface may be defined by many uninterpretable features, a single point in that decision surface can be locally approximated. The most popular implementation of this approach is Local Interpretable Model-agnostic Explanations

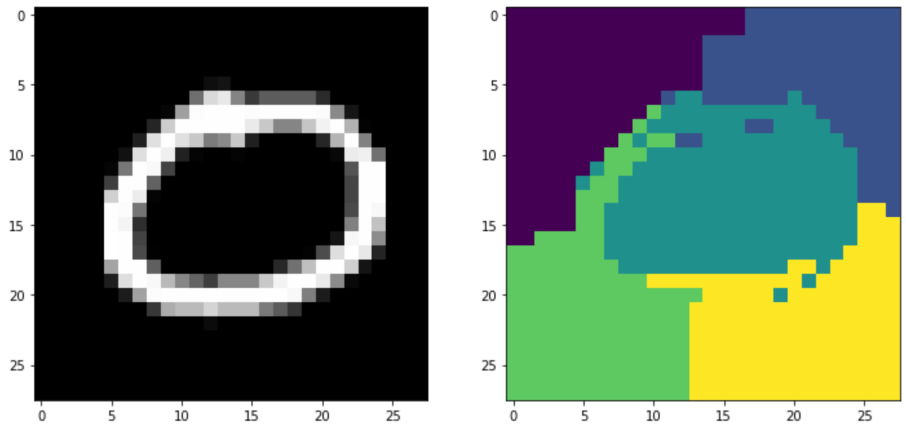

Figure 2.3: Example of LOO segmentation.

(LIME)[14]. LIME fits a flat plane against the decision surface, and infers the boundary by perturbing the input point to generate a cloud of points which are then all inferred by the model. While not being perfectly accurate, this is effective at broadly showing the explanation of the point inference and has the advantage of being model-agnostic. LIME still suffers from some the same limitations of LOO: each explanation requires multiple inferences (due to the input point perturbation), and to work over large dimensional data (such as images) superpixel segmentation must be used.

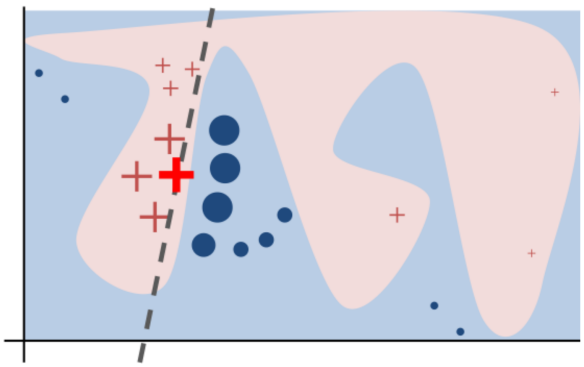

Figure 2.4: LIME local model. From Ribeiro et. al.[14].

2.2.4 Neural Network Approaches

A final approach is specific to neural networks: a series of techniques are available to compute the gradient of the class score with respect to the input pixels, yielding an effective explanation of which inputs are salient to the model's ultimate classification[23]. These methods are typically white-box, and operate over the specific structure of the neural network; this means that explanation method implementations need to be aware of the implementation details of the networks they are explaining.

There are a several methods of achieving this, with the simplest being Gradient Ascent[13]. In gradient ascent, the gradient of the class label with respect to each input feature is computed via backpropagation all in one go, leading to a per-feature im-

portance score across the image[23]. However, this can cause a very grainy saliency map (see Figure 2.5). A more nuanced approach is Deep Learning Important FeaTures (DeepLIFT), which compares the activations of network neurons to a "reference activation" (activations on a different instance) and can separate positive and negative contributions to give higher-quality results than Gradient Ascent[23, 19].

Figure 2.5: Example saliency map generated with gradient ascent, depicting the saliency map from the top class prediction from a ConvNet. From Simonyan et. al.[13].

A popular library for generating these explanations is Shapley Additive Explanations (SHAP)[15]. SHAP represents explanations as a set of Shapley values (a measure of how important a contribution is to an overall whole borrowed from game theory) of the overall model, and computes these using a unification of several techniques (notably LIME and DeepLIFT)[15]. This approach generates high-quality explanations and is packaged into an easy-to-use Python library `shap`[27].

SHAP is used heavily throughout this project, so it is worth examining its underlying principles of operation. SHAP generates explanations for each label for each instance; for example for each numeral in MNIST SHAP generates ten sets of explanations: one for label `0`, one for label `1`, and so on. This is visible in Figure 2.2, where each instance (rows) has ten different Shapley value sets (columns). These Shapley values are not necessarily the same magnitude between data points (though in this project they usually are), and are typically similar within a dataset. SHAP operates over not only image data (with DeepSHAP) but works with other types of data, such as tabular data—see Figure 2.6 for an example of an XGBoost classifier explanation for specific

labels over the Iris dataset.

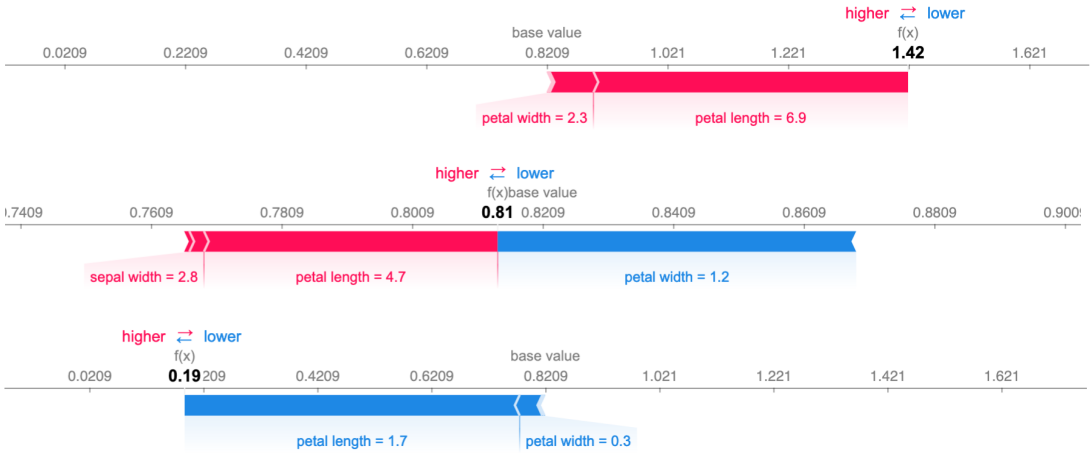

Figure 2.6: Force plots of SHAP explanations of XGBoost classifications trained on the Iris dataset, one for each class in the dataset. Note how each feature contributes different amounts towards the classification, with the final classification being a sum of the contributions.

2.3 Dimensionality Reduction and Hierarchical Clustering

2.3.1 DBSCAN

Hierarchical clustering algorithms are a series of algorithms to form clusters over data where the number of target clusters is not necessarily known; contrasting against partitioning algorithms (such as K-means clustering) which require a knowledge of the cluster count to partition the dataset. Hierarchical clustering algorithms work by combining data points into clusters agglomeratively, and this property makes them incredibly powerful for analyzing unknown data as they are able to discover the data[5].

A commonly-used clustering technique is Density Based Spatial Clustering of Applications with Noise (DBSCAN). DBSCAN is able to efficiently handle clustering over a dataset with arbitrary-shaped clusters, and is able to achieve this with minimal required knowledge of the underlying dataset (as it is parameterized by a single hyperparameter)[5]. This is achieved by picking an arbitrary point in the dataset and building a list of points reachable within a distance ε from that point (measured via a Euclidean distance metric). If that list is over a threshold number of points (typically 5), the list of points is given a cluster label—otherwise, it is marked as noise. This process is repeated until all points are either assigned a cluster or marked as noise.

However, due to DBSCAN's reliance on a Euclidean distance metrics, it is not suitable to very high-dimensional data. This is due to *curse of dimensionality*, which states that as the dimensionality goes up the more sparse the input space becomes for nearest-neighbor searching[3].

2.3.2 Principal Component Analysis

Over large datasets, the number of dimensions per each instance can become cumbersome to work with. To counter this, dimensionality reduction techniques can be used to reduce the number of dimensions for each instance. Principal Component Analysis (PCA) is a technique used to reduce the dimensionality of a dataset by finding a set of vectors (*principal components*) which form a basis for a new space in which the dataset can be transformed into[9]. These principal components are orthogonal to each other, and maximize the variance for the data. Figure 2.7 shows an example of what these vectors look like for the MNIST dataset. Principal components can be found by calculating the eigenvectors of the covariance matrix for the data, with the eigenvectors with the largest corresponding eigenvalues being the components with the maximum explained variance for the dataset[22]. PCA is sensitive to outliers, however, as these outliers can greatly increase the variance and skew the maximum variance for a particular component[22].

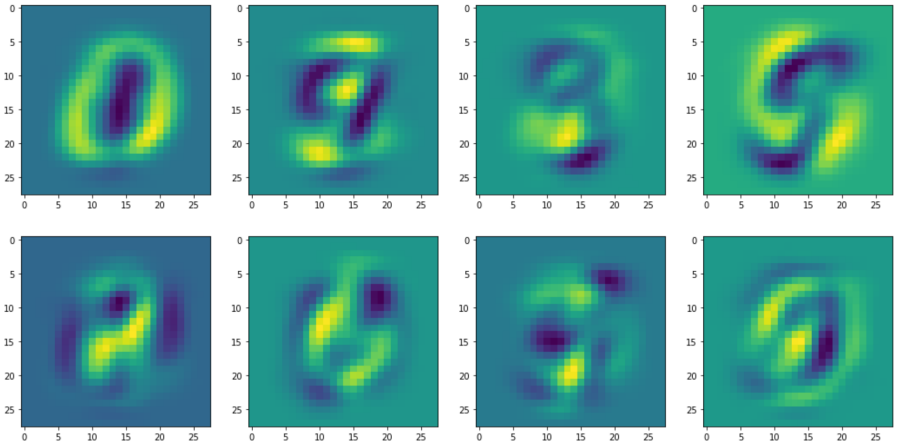

Figure 2.7: Example PCA vectors for MNIST.

2.4 Latent Structure Detection

Class introspection is a novel technique, but the idea of identifying latent classes within a dataset is not new. A prominent application area of this concept is medical studies, where similarities between subgroups of patients need to be identified in order to determine the safety of a drug; for example, if certain members of the experimental group had different reactions to a research pharmaceutical, then it would be important to both identify fragmentation inside that group (i.e. had a reaction to a drug in different ways) and to identify common features between those individuals (e.g. similar ages, etc.)[12]. Discovering these similarities is crucial, as treatment techniques can rely on a specific confounding variable in a population that is not readily captured by the available features.

2.4.1 Subgroup Analysis

Subgroup analysis is a simple method of detecting latent structure. Subgroup analyses are performed by manually categorizing gathered data into subgroups based on some common characteristic of the data, and are typically examined by incorporating some moderating variable into a regression and interpreting the results[8]. However, because this grouping is inherently observational, this method is difficult to use effectively and is time-intensive. Additionally, the results of such analyses are subject to high Type I errors, as false positives are easy to make when manually grouping data[12].

2.4.2 Finite Mixture Models

Finite mixture models are another way of determining latent structure (or structure in general)[12]. Finite mixture models represent the data as a linear combination of component densities and maximize the likelihood of a specific configuration of models, and a common density function to use is the Gaussian normal distribution[24]. Gaussian mixture models optimize the probability:

$$p(x) = \sum_{m=1}^{M} P(m)\mathcal{N}(x;\mu_m,\Sigma_m)$$

where the probability $p(x)$ of a specific point is given by a sum of normal distributions moderated by mixing parameters $P(m)$[24]. This probability is typically optimized by the expectation-maximization (EM) algorithm[11].

Finite mixture models can find latent structure, but they are limited in their effectiveness for this task. Finite mixture models are limited to discovering structure in the form of the basis functions chosen (e.g. Gaussians). This may not be enough statistical power to identify latent structure in complex data by itself, but if combined with a explanatory technique it may be enough to interpret saliency maps.

2.4.3 Commonality metrics

A cutting edge technique in this area is the detection of rare subclasses via *commonality metrics*[18]. This technique analyzes the input training classes for a given dataset and for each class determines the average activation in the penultimate neural layer for that class, and then scores each instance based on how similar it is to that average activation.

This is technique does not identify latent classes in and of themselves; rather, it identifies single instances that may be mislabeled or are significantly different than the average. Additionally, because the commonality metric approach operates over an average activation, large numbers of far-from-average instances will skew the whole commonality metric—potentially rendering the method less effective at identifying singular anomalous instances.

Chapter 3

Methodology

3.1 Problem Formulation

The central problem to be solved is the discovery of latent subclasses in the input space. The aim is to find unlabeled (latent) subclasses within labeled classes in an existing dataset, with the hypothesis that these latent subclasses can carry additional information that is relevant to the authors of the classifier (see the examples in Section 1.1). The latent subclasses should be differentiated from one another without human intervention; that is to say, this should be an unsupervised technique.

This is, at its core, a clustering problem. In the ideal case (with no latent structure), each discovered cluster should correspond to a single true label in the dataset. However, in the case where latent subclasses exist within a class, we expect to see two or more cluster labels assigned to a single class label. In general, this method should take a dataset as input and the labels and output a list of cluster labels for each instance.

3.2 Method Overview

Class introspection's central hypothesis is that even if a instances within a dataset is all labeled identically, the specifics of how a specific classifier *interprets* that instance can change significantly between those instances—and it is precisely that change that can be used to determine the existence of latent subclasses. In other words, this approach solves the subclass detection problem by leveraging an existing classifier, and through that the classifier's statistical power. Detecting these instance changes relies on the combination of several methods discussed in Chapter 2. The general process is described below:

1. Generate classifications for all instances in the dataset.
2. For each instance, create a local explanation of the classifier's decisions.
3. For each class:
    (a) Select all instances in the dataset classified as the selected class, and their

explanations.

(b) Run a clustering algorithm over the class's explanation.

(c) **Multiple clusters indicate the existence of latent structure.**

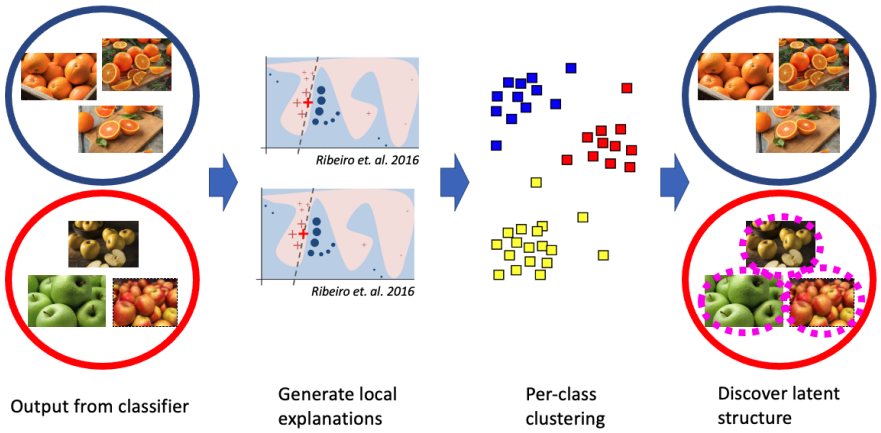

Figure 3.1: Class introspection methodology overview.

Clustering over the instance explanations is possible as these explanations act as a *proxy* for the underlying instance. The explainability methods discussed in Section 2.2 produce saliency maps over the input features[14, 15], and these saliencies are expressed as a tensor of saliency weights over the initial features. This is crucial: because these saliency maps are numeric, they can be interpreted just as easily as the original data—or even more easily, as discussed further on in this paper.

Because the explanations are simply numeric they can be clustered over, but the clustering itself is a challenge. Because the total class count is unknown, partitioning algorithms (e.g. K-means clustering) are not suitable to the task—if the total class count was known, the problem would be solved as we'd already know which subclasses exist! Instead, agglomerative clustering methods are appropriate for this use case, as they can operate over an unknown number of clusters. In this paper, DBSCAN is used to apportion the explanations into clusters—although others can be used (see Section 6.3).

Before clustering can be applied, the dimensionality of the data must be reduced. Clustering algorithms rely on (typically Euclidean) distance metrics, and as the number of dimensions increases the number of unit cells in that space increases exponentially, and so in these high dimensional spaces the nearest point can be extremely far away[10]. In this paper, principal component analysis is used to reduce the number of dimensions before clustering, yielding much better results.

At the end of the clustering process, the assigned cluster labels within each class are displayed as a histogram. The key insight here is that the cluster labels correspond to the similarity of the input images to each other, and if there are a large number of instances in two or more clusters those high-count are a candidate for a latent subclass.

Ultimately, this method requires human interpretation of the results as some of the

detected latent structure may be intentional (e.g. in the apples-oranges example the classifier may not care about the different kinds of apple). Even so, the output from this algorithm is much more readily interpretable than attempting to audit the dataset and labels by hand.

3.3 Baseline

A simple baseline technique for detecting this latent structure is to ignore the explanatory methods altogether and perform clustering over the raw instance data itself. PCA is used to reduce the dimensionality of the data before being passed to the clustering algorithm, DBSCAN. This approach is similar to the commonality metrics methods described in Section 2.4.3, and is nearly identical to the methodology in Figure 3.1 (see Figure 3.2 for more details). To generate a dataset with latent structure, artificial latent structure is induced as in Section 4.1.

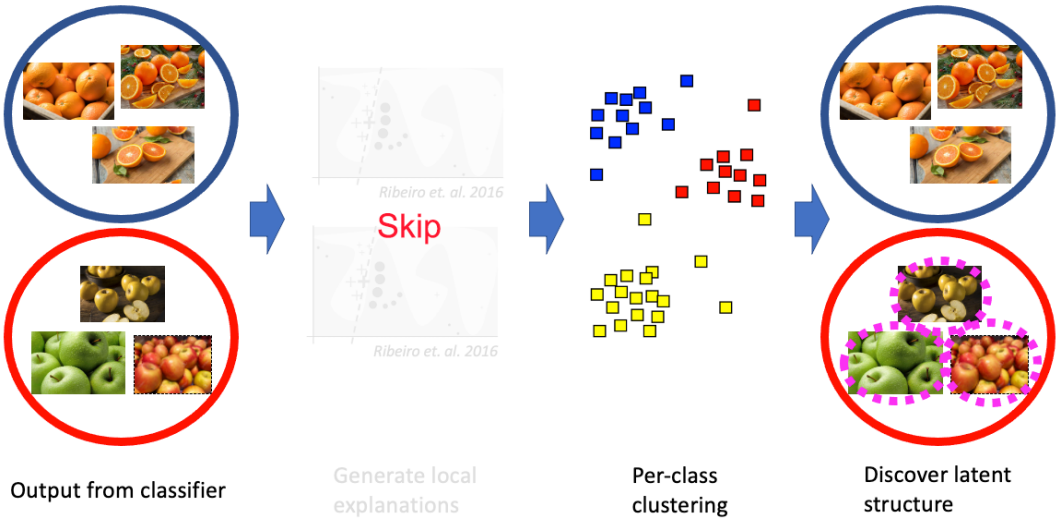

Figure 3.2: Overview of the baseline pipeline configuration.

Running the baseline over the MNIST dataset yields a surprising result: the PCA and DBSCAN over the raw MNIST digits is not likely to determine when the artificial latent structure is present! Even with a hyperparameter sweep of the DBSCAN ε parameter, in some cases an optimal configuration cannot be found where the bridged class has a significant split in cluster membership as opposed to the non-bridged classes. Figure 3.3 contains the output for the $(1 \rightarrow 8)$ bridging (identical to the bridging in Figure 4.21), and it is clear that the bridged class is not identified: no class contains any split in cluster membership, and some are entirely noise. However, some splits do work: in the case of the $0 \rightarrow 1$ split in Figure 3.4, the latent structure is successfully identified.

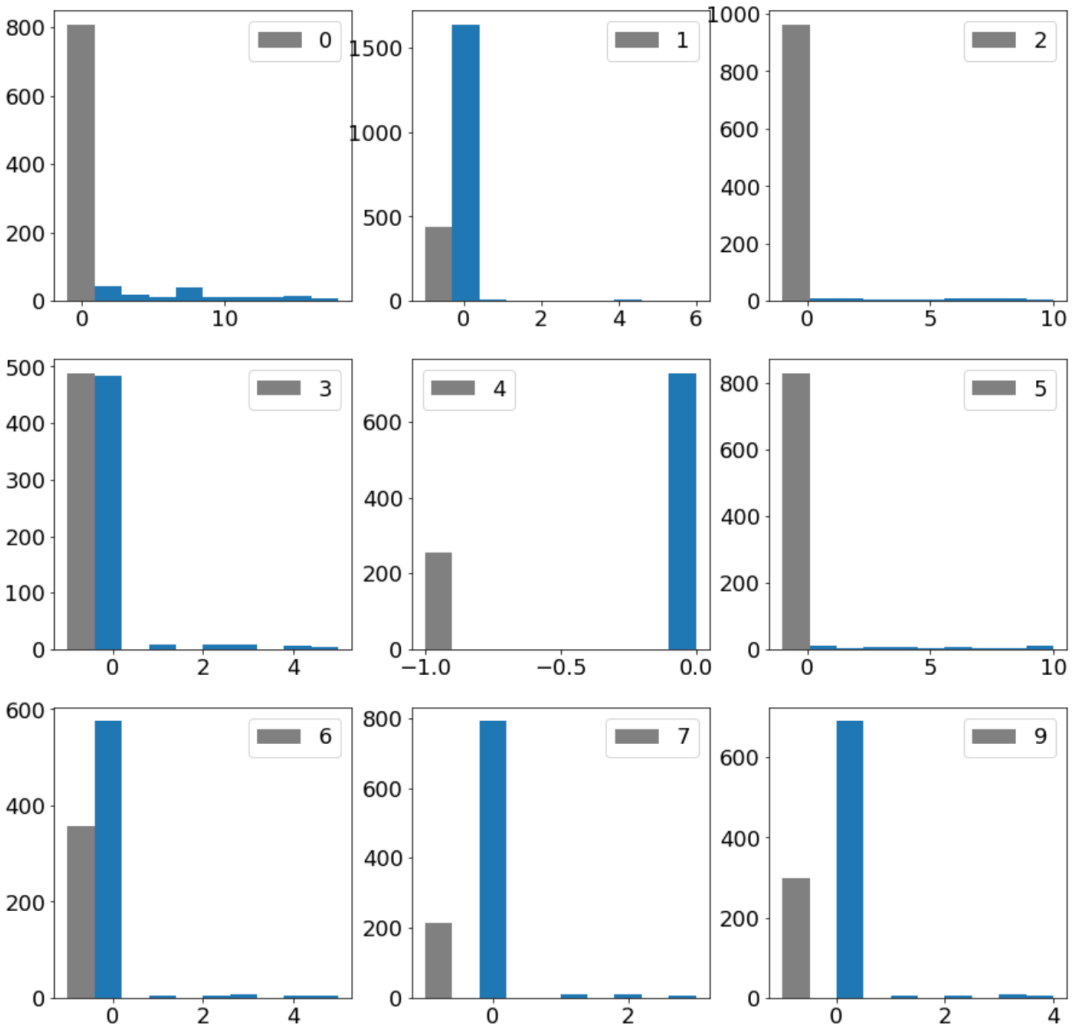

Figure 3.3: Detected class fragmentation from the baseline run against MNIST $1 \rightarrow 8$ with $\varepsilon = 250$. Note how none of the classes show a major split in the assigned cluster labels (ignoring gray noise bars)

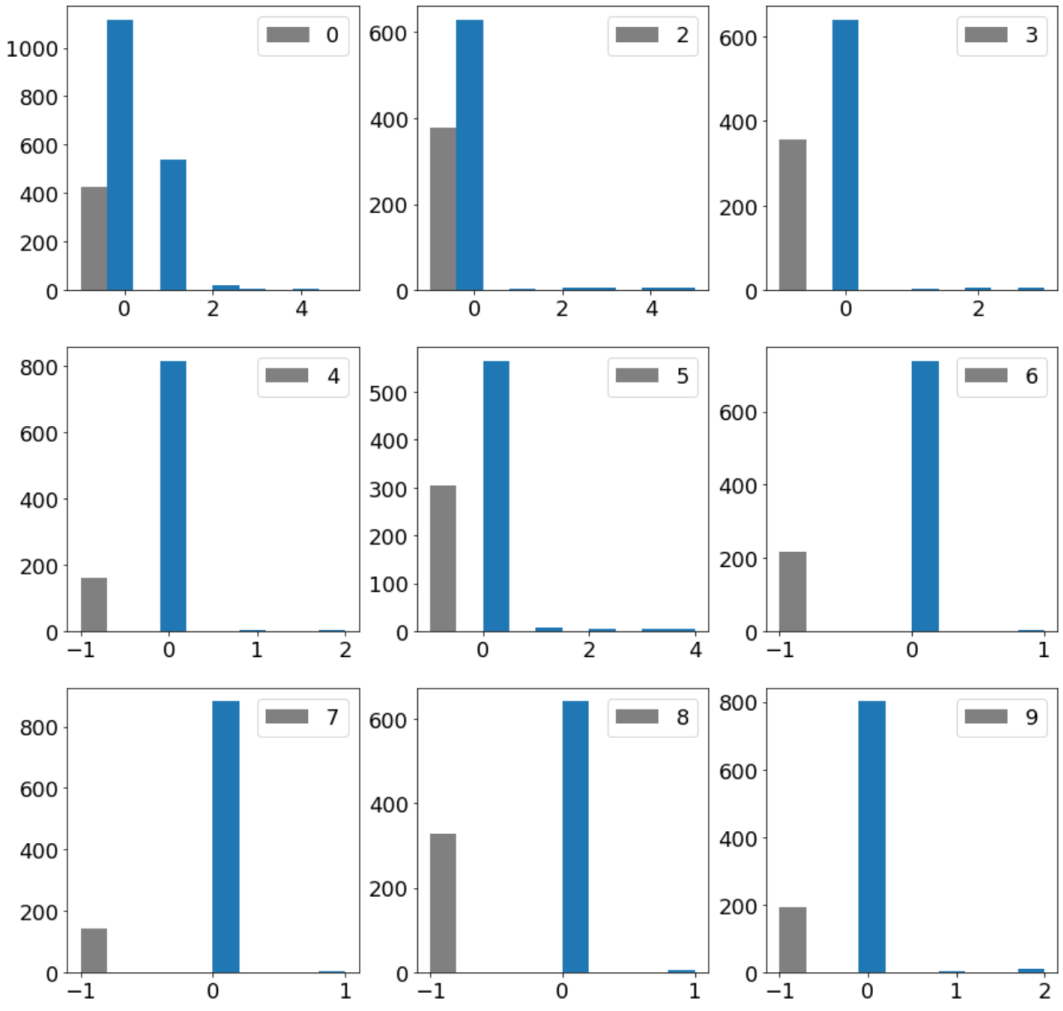

Figure 3.4: Detected class fragmentation from the baseline run against MNIST $0 \rightarrow 1$ with $\varepsilon = 275$. Note how here, the latent structure is successfully identified (top left).

Chapter 4

Initial Explorations

4.1 Artificial Latent Structure

To develop a pipeline to detect latent structure, we first need a dataset which contains latent structure. Unfortunately, it is difficult to find a dataset with such structure already in it. The challenge is twofold—either the dataset is simple enough that latent structure is virtually nonexistent, or the dataset is complex enough that detecting latent structure requires domain knowledge to detect. For a useful comparison of class introspection techniques, the dataset needs both the original class labels and labels for the latent structure to be detected. Creating these labels requires manually labeling every instance, which is both incredibly time-consuming and out of the scope of this project.

Clearly, another solution is needed. Instead of relying on existing datasets to have exploitable latent structure, we can *create* latent structure ourselves by "bridging" class labels together. This trivially creates latent structure, as both the instances for label A and label B are bound to the same class—and thus the new bridged class label has two distinct types of instance in it. This also has the benefit of giving labels corresponding to the original classes, as the original labels can be compared to the bridged labels to show which instances have been bridged. See Figure 4.1 for an example with MNIST with labels `1` and `8` bridged together.

This approach has limitations, however. The primary complication is the nature of the latent structure itself: bridging two classes can create a superclass with extremely obvious latent structure, which may not align well with real-life latent structure. However, for the purposes of this project these effects have been ignored, as it is useful to have obvious latent structure to detect rather than very subtle structure in order to test the pipeline. Additionally, bridging class labels can lead to unexpected behavior whilst training models over a dataset with a small number of classes. This is a fundamental issue, as bridging two classes effectively removes one of the classes. For simple datasets, this can cause certain models to not learn the structure of the superclass but instead simply learn one of the other classes and segment the input space into "class and not class". This was encountered during experimentation with simple datasets, discussed below in Section 4.2.3.

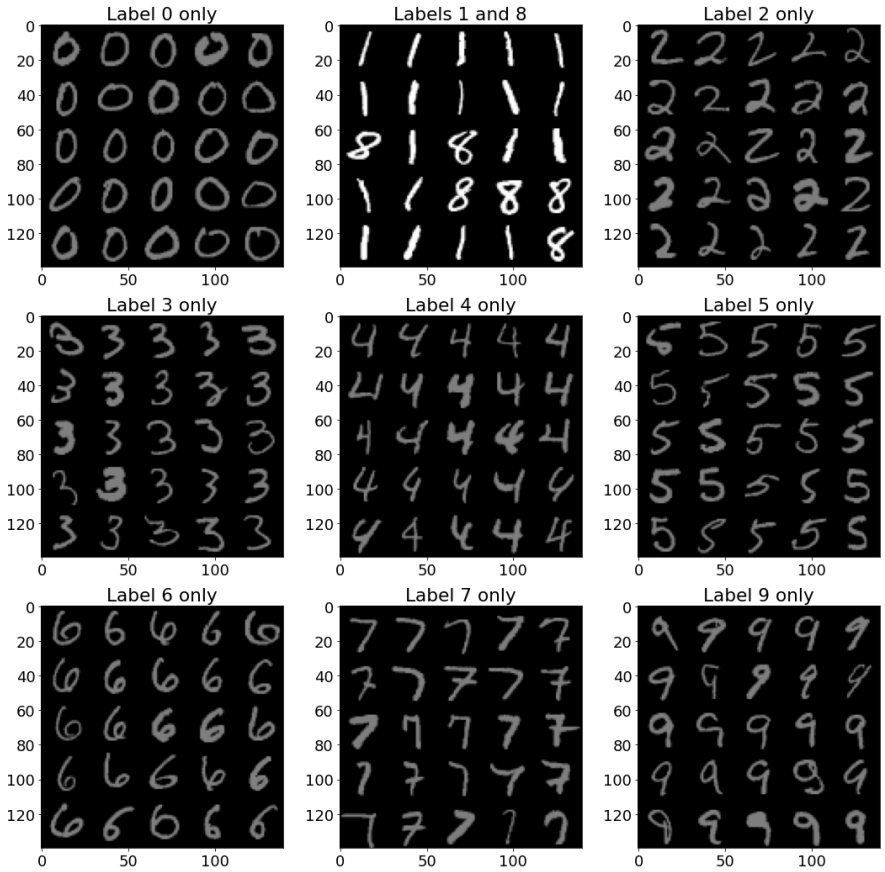

Figure 4.1: A demonstration of latent structure with MNIST, here bridging the `1` and `8` classes. Note that the bridged classes have been highlighted.

4.2 Iris plants dataset

The initial proof-of-concept of the class introspection pipeline needed to be as simple as possible, so the Iris dataset was chosen. The Iris dataset is a simple dataset, consisting of three classes characterized by four features. The petal width, petal length, sepal width, and sepal length are measured for three different species of iris; *Iris setosa*, *Iris virginica*, and *Iris versicolor*[2, 1]. Based on these four features, it is possible to determine the class of an instance—and in fact, the petal length and width alone are highly correlated with the class.

Due to the relative simplicity of this dataset, it is a common dataset for multiclass classification scenarios. For the initial proof-of-concept, it was chosen specifically for its low class number and distinct class separation. With the petal width and petal length correlated highly with class and sepal width and sepal length not correlated, it is possible to ignore sepal attributes and only consider two dimensions for plotting—allowing for easy debugging of the pipeline.

Figure 4.2: *Iris versicolor*[6].

4.2.1 Non-bridged Performance

The first experiment was simply to generate explanations for each instance using simple models, and plot them against each other to see if any structure emerged. The models used in the analysis were off-the-shelf Scikit-learn models, using black-box explanations provided by LIME. All experiments used the same training/testing split of 25%.

The first model tested was a random forest ensemble model (`sklearn.ensemble.RandomForest`) with $n = 15$ estimators. Training the model resulted in a perfect score against the test data, indicating a potential overfit; however, for class introspection overfit is not of particular concern. Figure 4.3 shows the results from this analysis. The graph shows that while the *Iris versicolor* instances are largely explained similarly, the *Iris virginica* and *Iris setosa* instances have some strange structure to the explanations—they are not stable within the class. Note that only the intra-class positions of the points matters, not the positions relative to other classes.

The next experiment used a multinomial naïve Bayes classifier (`sklearn.naive_bayes.MultinomialNB`). This was trained exactly in the same manner as the random forest ensemble, and again fit well to the data. The graph (Figure 4.4) again shows a tight clustering for the *Iris versicolor* and more spread out explanations for *Iris virginica* and *Iris setosa*, though this time with different locations of the classes relative to each other. While this does not affect anything from a class introspection perspective, it is noteworthy that while the explanations are in different locations the overall class shapes are similar between the multinomial naïve Bayes classifier and the random forest classifier.

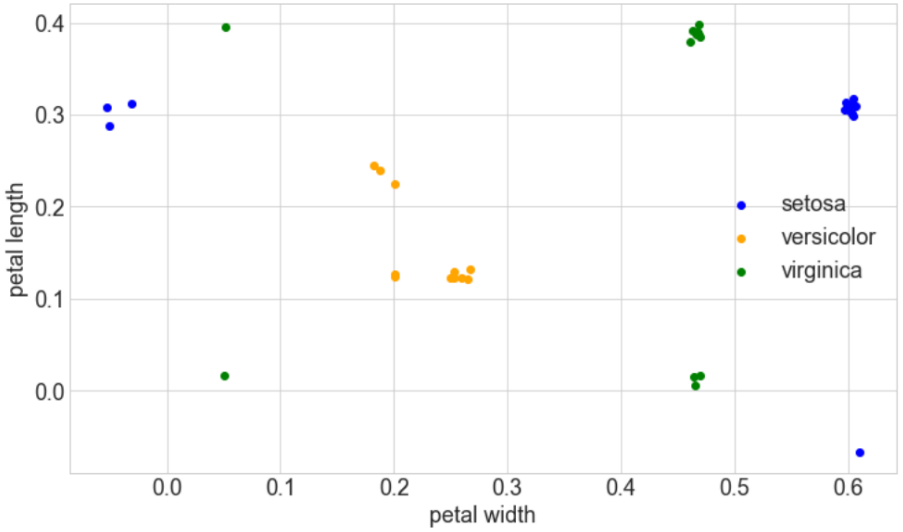

Figure 4.3: LIME explanations of random forest classifier over all irises (petal features only).

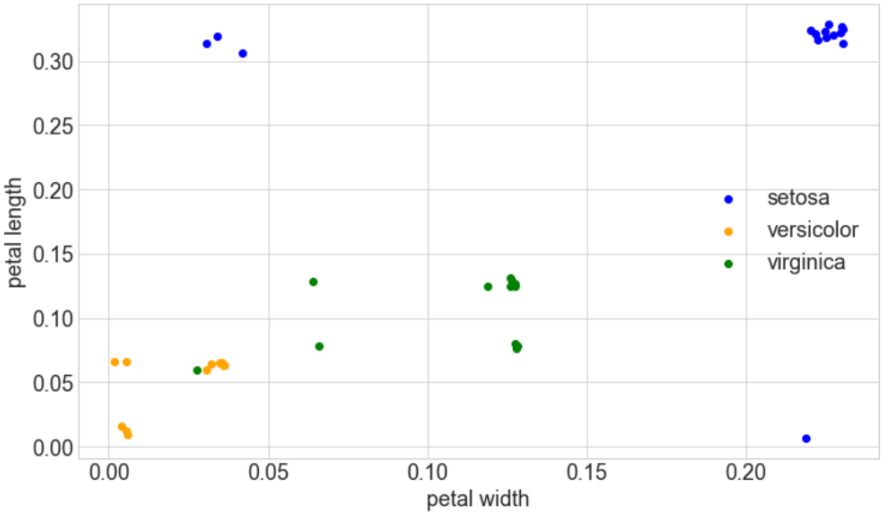

Figure 4.4: LIME explanations of multinomial naïve Bayes over all irises (petal features only).

4.2.2 SHAP vs LIME

These experiments uncovered an issue with LIME—the explanations within each class were not clustered together, but rather exhibited a strange four-sided symmetry. Because the dataset is fairly simple, it was reasonable to assume this to be a limitation in LIME.

To combat this, SHAP was used instead to generate explanations. The experimental setup was similar, with a tree-based model (XGBoost with 100 rounds and a learning rate of 0.01) used instead of Scikit-Learn's random forest classifier. This resulted in better performance the Iris dataset (see Figure 4.5), with the majority of instances within classes having explanations within the same cluster. There are a few outliers,

for example within the *Iris versicolor* cluster which has a singular outlier. This shows SHAP's improved performance over LIME, as the explanations are much higher quality.

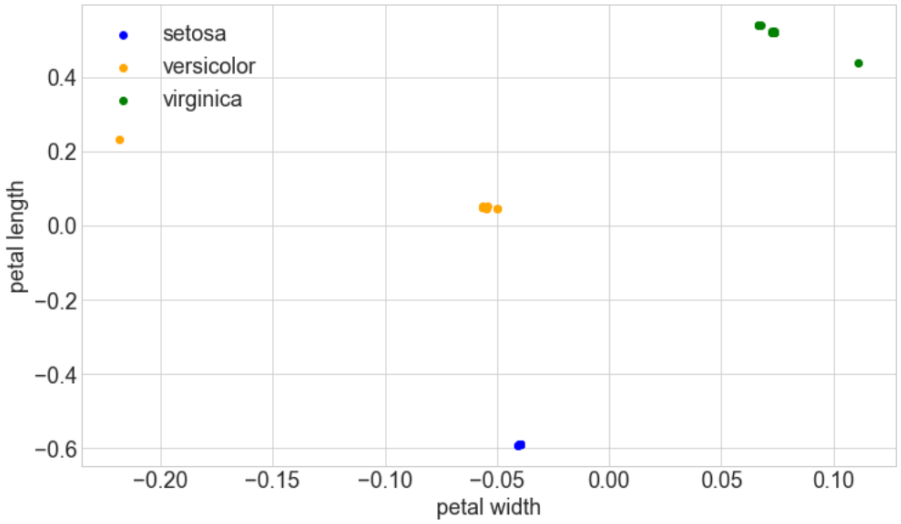

Figure 4.5: SHAP explanations of XGBoost over all irises (petal features only).

4.2.3 Bridged Performance

With this improved performance of SHAP powering the explanations, the next experimental step was to introduce latent structure into the dataset as described in Section 4.1. For this run the *Iris setosa* and *Iris versicolor* were bridged together and the resulting dataset was trained as before, with an XGBoost classfier with 100 rounds and a learning rate of 0.01 and explanations generated via SHAP's tree explainer.

Unfortunately, this exposed the issue with class bridging with low-class-count datasets: instead of learning the differences between the instances in the bridged class, the tree classifier simply split the inputs into two categories: *Iris virginica* and not *Iris virginica*. Figure 4.6 shows the full bridged class explanations, showing a split inside of the bridged class. However, further investigation of this split shows that the *Iris versicolor* and *Iris setosa* classes are not separate from each other within the class (see Figure 4.7).

4.3 MNIST dataset

Based on the above experimentation, the Iris dataset's low class count made it an unsuitable choice for introducing latent structure and thus an unsuitable choice for developing the class introspection algorithm against. A much better multiclass dataset is the MNIST dataset[7] which contains $70,000$ handwritten digits (0 through 9) as 28 pixel by 28 pixel monochrome images (see Figure 4.8). The MNIST dataset is a common dataset in machine learning as it is easily understood by beginners while still being complex enough to allow for meaningful analysis.

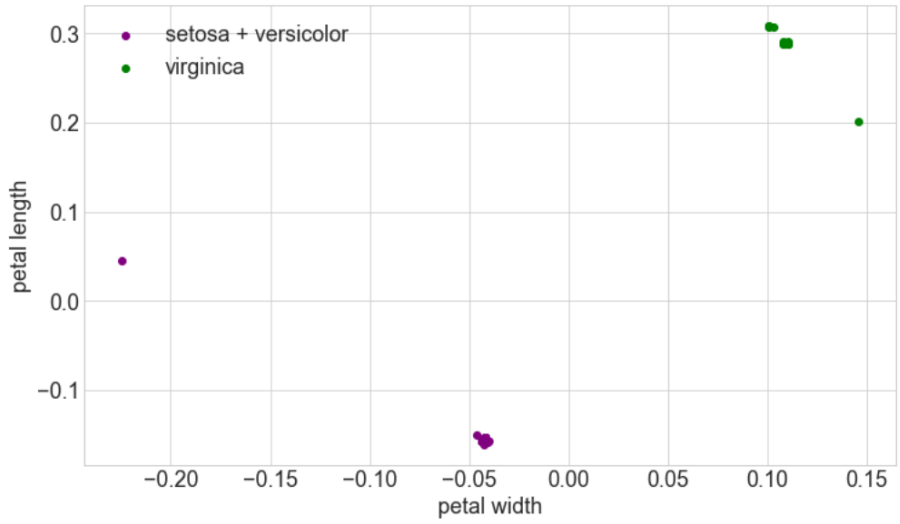

Figure 4.6: SHAP explanations of XGBoost over irises with bridged *Iris setosa* and *Iris versicolor* classes (petal features only).

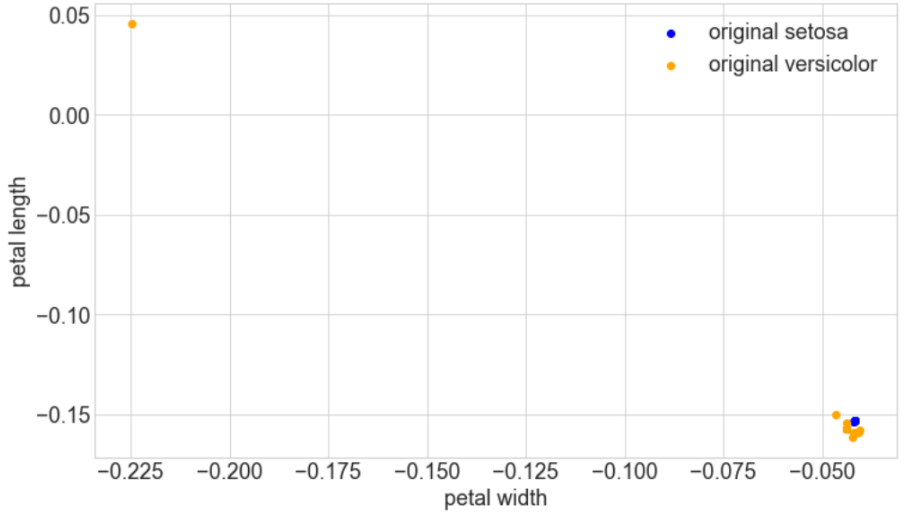

Figure 4.7: SHAP explanations of XGBoost over only bridged *Iris setosa* and *Iris versicolor* classes (petal features only).

For the purposes of class introspection, it is an ideal dataset due to its relative simplicity and high class count. This high class count is crucial to avoid the issues with too few meaningful classes discussed above, as with a single bridged class there are still eight remaining classes.

4.3.1 LIME and PyTorch

Because MNIST is image data, the models used for the tabular Iris dataset would not have been sufficient to classify MNIST digits. Instead, an artificial neural network (ANN) was used as a classifier, with three fully-connected hidden layers of $784 \rightarrow 128 \rightarrow 128 \rightarrow 64 \rightarrow 10$ neurons connected by rectified linear units (ReLUs) with a

Figure 4.8: A sample of MNIST digits.

logarithmic softmax on the output (see Figure 4.9).

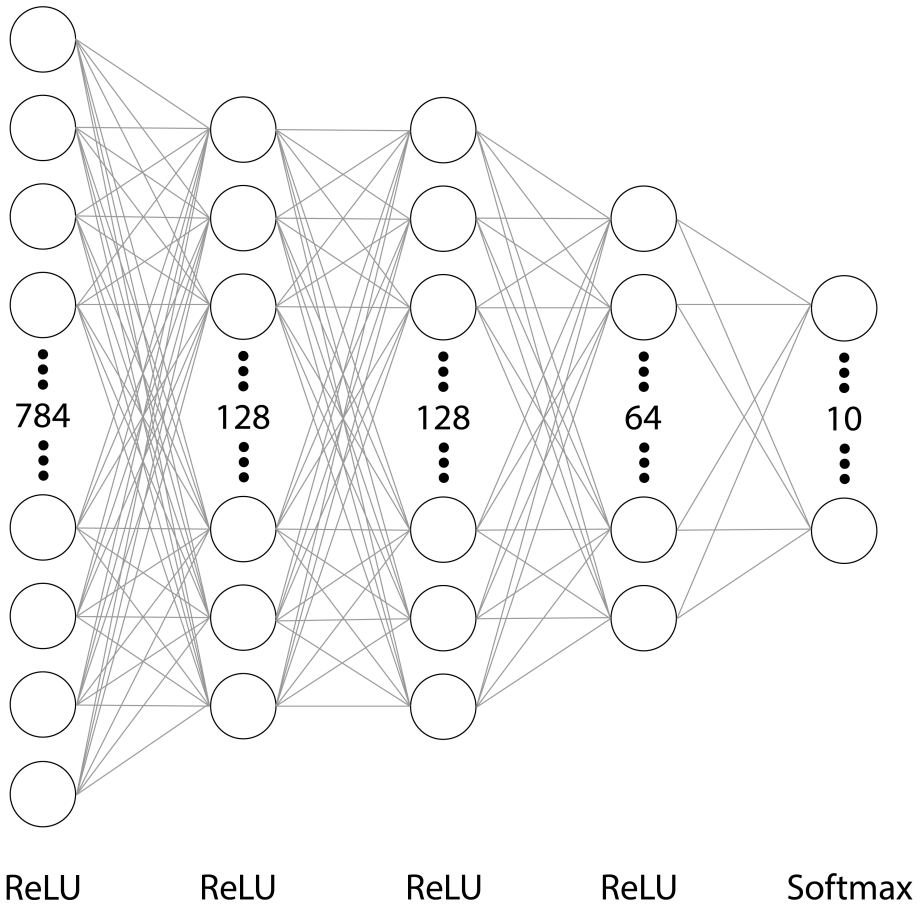

Figure 4.9: ANN architecture for MNIST detection.

For the initial MNIST experimentation, the ANN was implemented with PyTorch and explained with LIME. PyTorch was chosen because of it's simple integration with LIME's default image explanation pipeline. First, as a sanity check the network was trained on the unbridged MNIST data, and then the test data instances were explained in bulk. The train/test split was the MNIST recommended $60,000$ training instances and $10,000$ testing instances[7]. The network reached an accuracy of 97.6% over the test set. In Figure 4.10, the results of these explanations can be seen—the LIME explanations do highlight the structure of the instance as being salient to the classifier output, though this is noisy. In Figure 4.11, the output for a specific instance (a 7) is shown. Again, it's clear that the bulk of the digit is marked as salient, but a lot of the background is also marked as salient. This noise is discussed below.

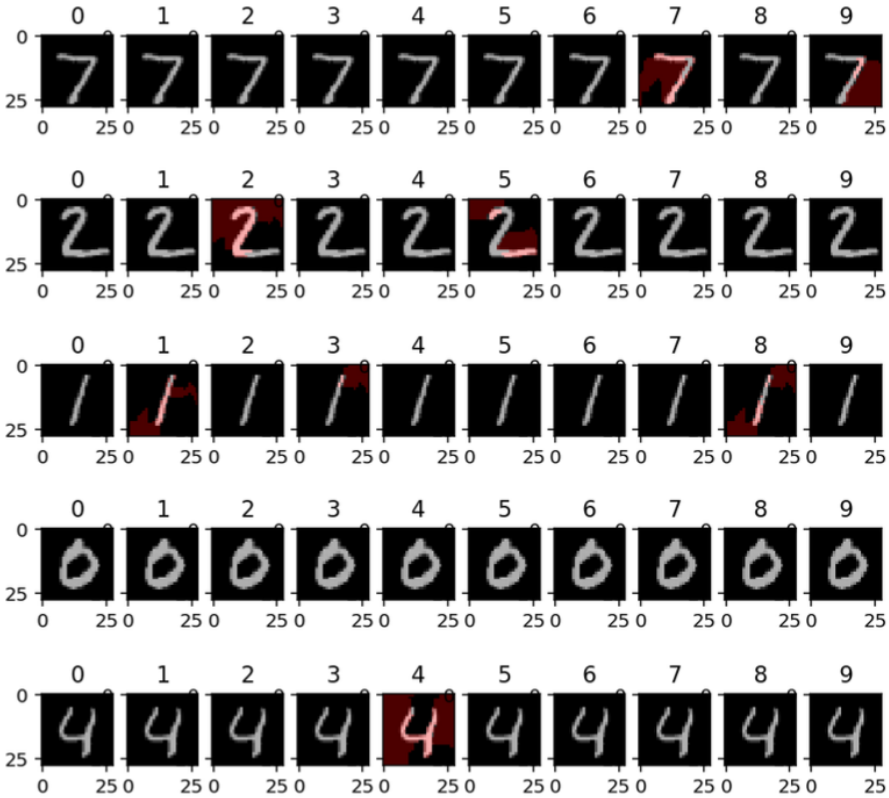

Figure 4.10: Unbridged MNIST explanations with LIME-generated (quickshift) superpixels.

The next step was to re-train the network with a pair of bridged labels. For this step, the 1 and 8 labels were chosen for bridging, as these digits are very distinct from one another giving the class introspection the best chance of succeeding. The labels were modified for both the train and test datasets, and the network was re-trained. In Figure 4.12, it is evident that the network marks both 1 and 8 as being in the same class.

While this is a positive result, this also exposes underlying issues with using LIME as an explainer. In image mode, LIME operates over superpixels generated by a generic segmentation algorithm (by default, scikit-image's Quickshift), running model inference over permuted superpixels rather than individual features for performance[14].

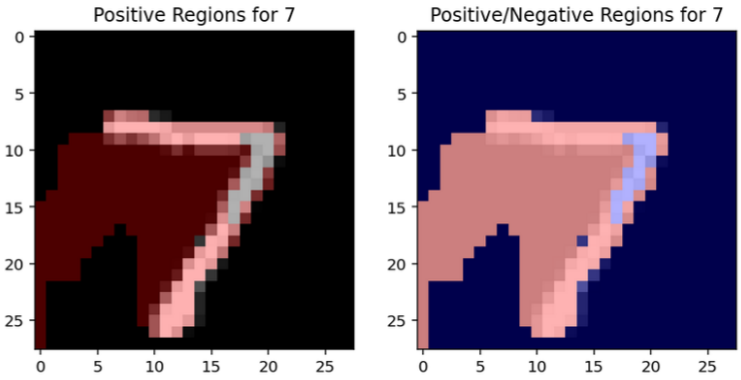

Figure 4.11: Unbridged MNIST explanations with LIME-generated (quickshift) superpixels, detail on individual instance.

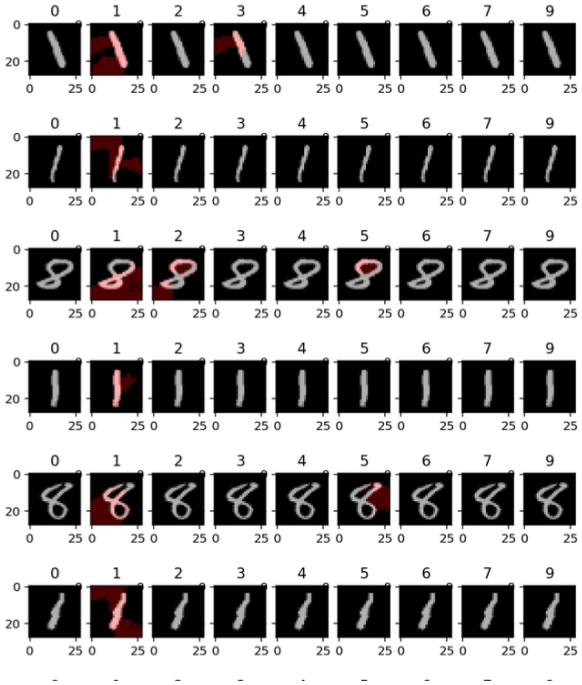

Figure 4.12: Bridged MNIST explanations with LIME superpixels.

However, using superpixels leads to very coarse-grained explanations, as there are only a small number of superpixels by default—see Figure 4.13 for a detail of a singular instance with very noisy explanations. This is not optimal for detecting latent structure, as more fine-grained feature importances are required.

4.3.2 SHAP and Keras

Due to the fine-grained explanations required for class introspection, LIME's explanations were insufficient. A different explanation engine was required, and SHAP (with the DeepLIFT backend) is able to provide those explanations. Instead of operating over superpixels, SHAP uses DeepLIFT to backpropagate the contributions of every

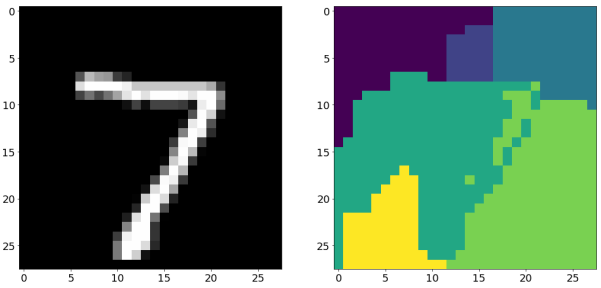

Figure 4.13: Detail of LIME superpixels.

neuron to every feature in the input space[19, 15].

SHAP is designed to work with Keras, so the previously-implemented PyTorch model would not be compatible with SHAP's image pipeline. Therefore, the model was re-implemented in Keras with the same specifications: fully-connected $784 \rightarrow 128 \rightarrow 128 \rightarrow 64 \rightarrow 10$ layers with ReLUs and a softmax activation (see Figure 4.14).

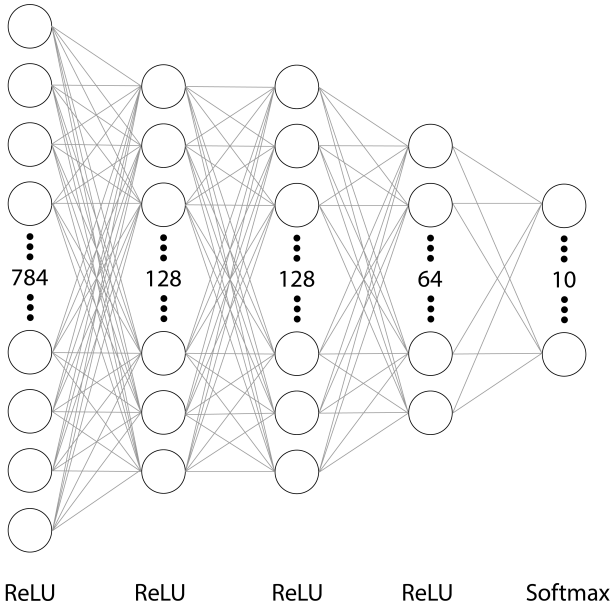

Figure 4.14: MNIST ANN implemented in Keras.

Training the Keras network resulted in a 97.91% accuracy over the test dataset, which is equivalent to the PyTorch model trained earlier. Running SHAP explanations over the test dataset yielded fine-grained results as shown in Figure 4.15. The structure of the explanations is much more evident with SHAP, as an example for an instance with a true label 7 the explanation for label 0 shows that the network expects a round shape, and because the instance does not match that shape the input features negatively contribute towards classification as a 0 (see Figure 4.16) For the same instance, the explanation of the true label 7 shows the shape of the 7 positively influencing the classification of a 7, yielding a classification of 7 for the instance (see Figure 4.17).

Again, the 1 and 8 digits were bridged together due to their dissimilarity, and the network was trained again (achieving an accuracy of 97.92%). Figure 4.18 shows

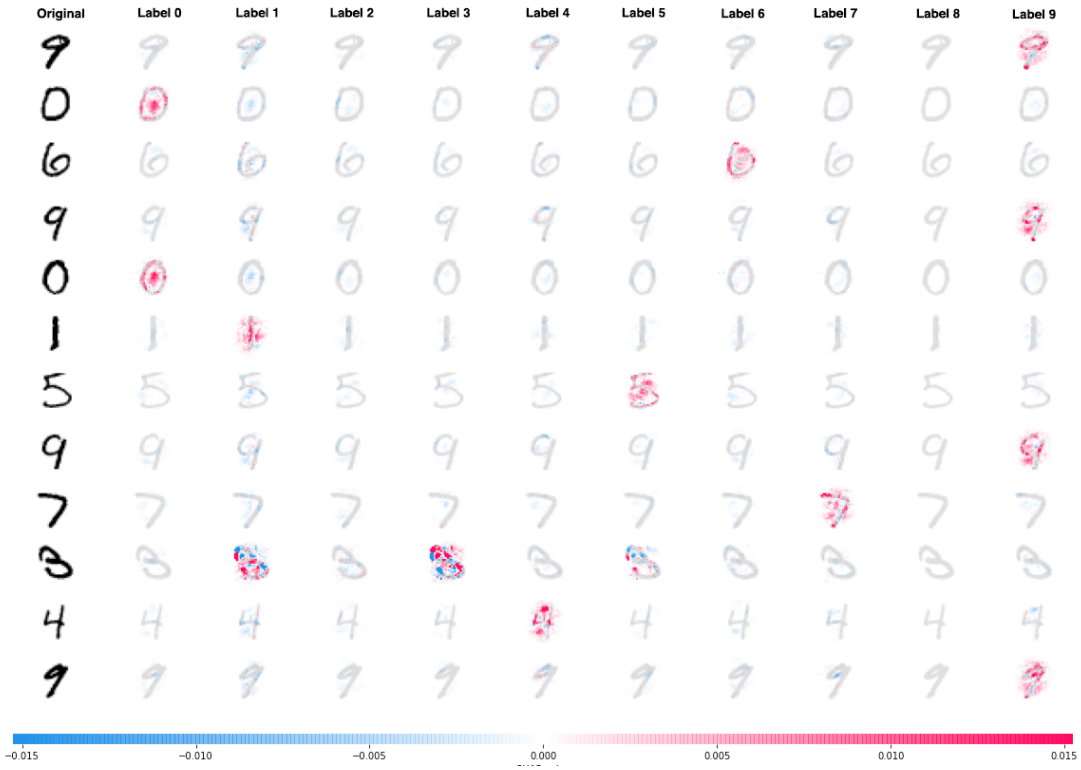

Figure 4.15: SHAP explanations over unbridged MNIST data.

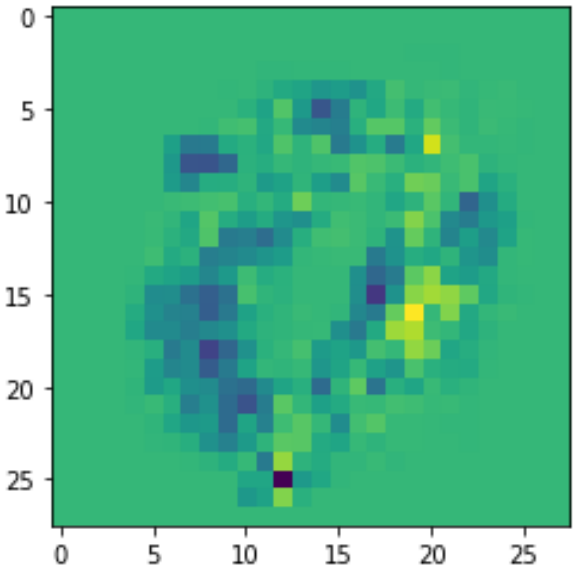

Figure 4.16: SHAP explanations over unbridged data, detail on explanation for label 0 and true label 7. Note the band of blue values in the shape of a zero, these are negative contributions (blue) from the features that would match label 0 and are missing on the 7.

another grid of SHAP explanations, note that there are no explanations in the 8 column as they have been bridged with the 1 column. Filtering just for instances in the bridged

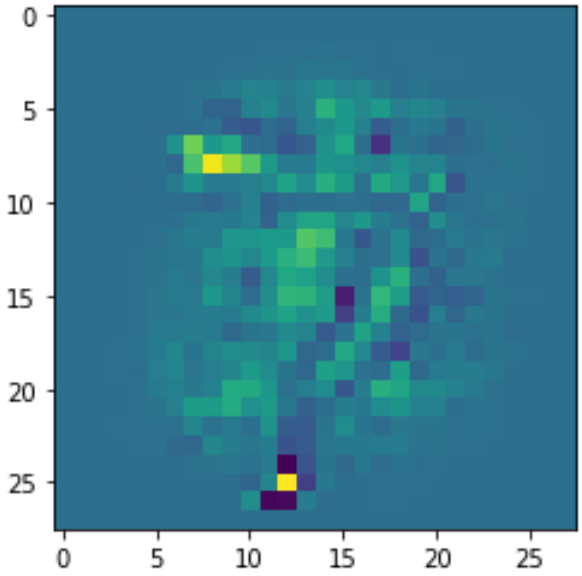

Figure 4.17: SHAP explanations over unbridged data, detail on explanation for label 7 and true label 7. Note the positive contributions (green) along the body of the glyph.

category shows a clear difference between the explanations for the 1 instances and the 8 instances, as seen clearly in Figure 4.19.

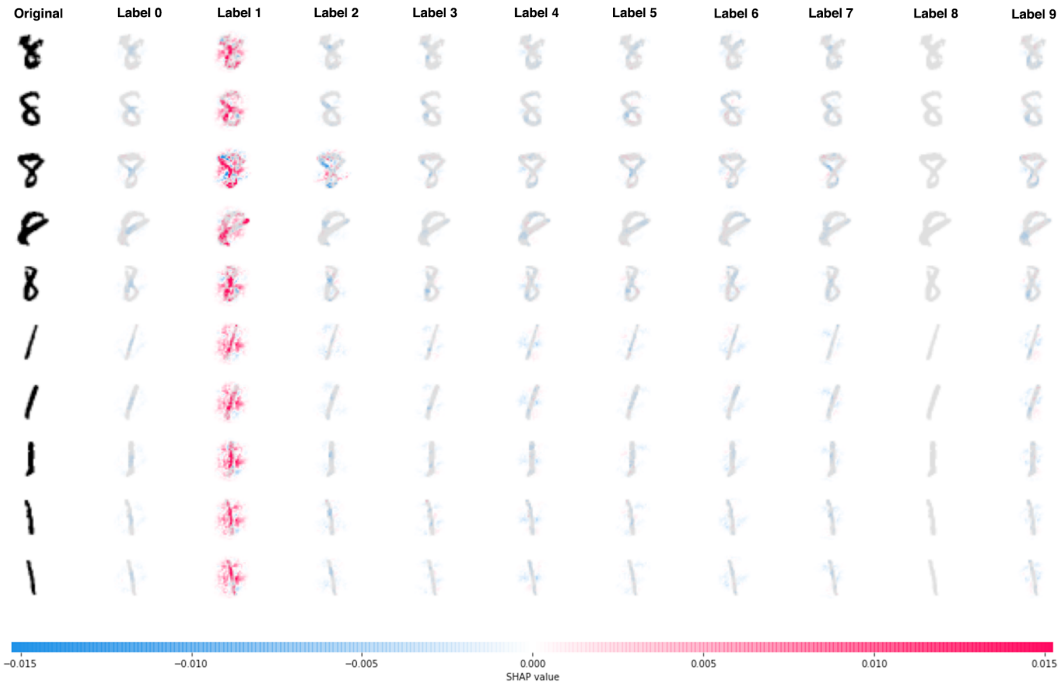

Figure 4.18: SHAP explanations over bridged MNIST data.

The next step in the class introspection pipeline is to cluster the explanation to isolate the latent structure in the class. This poses several problems: the number of clusters

Figure 4.19: SHAP explanations over bridged MNIST data, detail on bridged classes.

is unknown, the data may not be linearly separable, and the data is high-dimensional. To solve the unknown cluster count, hierarchical clustering is used (DBSCAN). DBSCAN can handle clusters that are not linearly separable, making it suitable for this application[5]. However, DBSCAN relies on a euclidean distance metric, requiring dimensionality reduction to avoid the curse of dimensionality.

In this case, PCA was used to reduce the dimensionality from 784 to 5 as a preprocessing step, yielding the principal components seen in Figure 4.20. Running DBSCAN over this data with $\varepsilon = 0.004$ yielded two distinct classes, visible in a histogram of cluster membership (Figure 4.21). This is a positive result!

To ensure that this is not a fluke, a different set of classifications (4) was chosen to run the pipeline over. Again, SHAP explanations were generated for all members of the class and PCA was applied. The first five PCA components were used as inputs to DBSCAN (again with $\varepsilon = 0.004$), and the class membership histogram shows that there is only one class with the similar settings (see Figure 4.22).

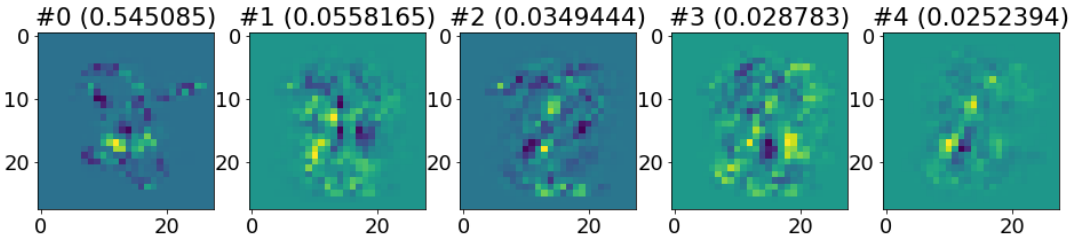

Figure 4.20: PCA vectors for the 1 to 8 bridge, with the explained variance of each component.

This shows that the it is possible to find latent structure with this method!

However, these results are not ideal. To improve performance, there are several steps

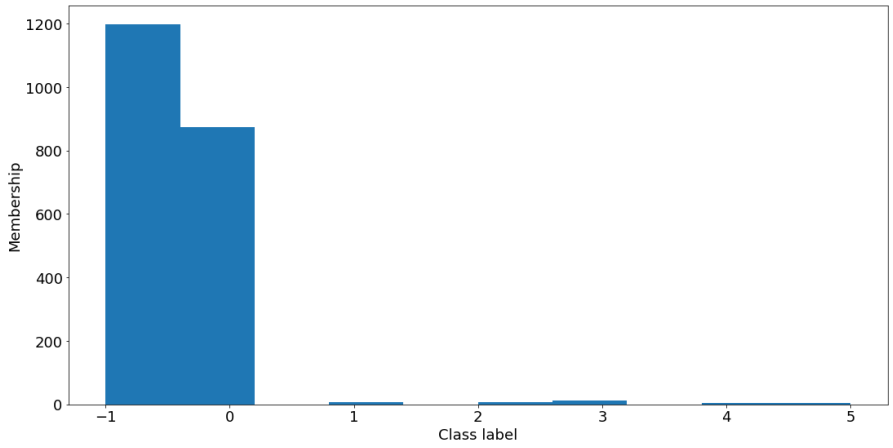

Figure 4.21: Intraclass fragmentation viewed with DBSCAN on bridged MNIST data ($1 \rightarrow 8$), shown as a class membership histogram.

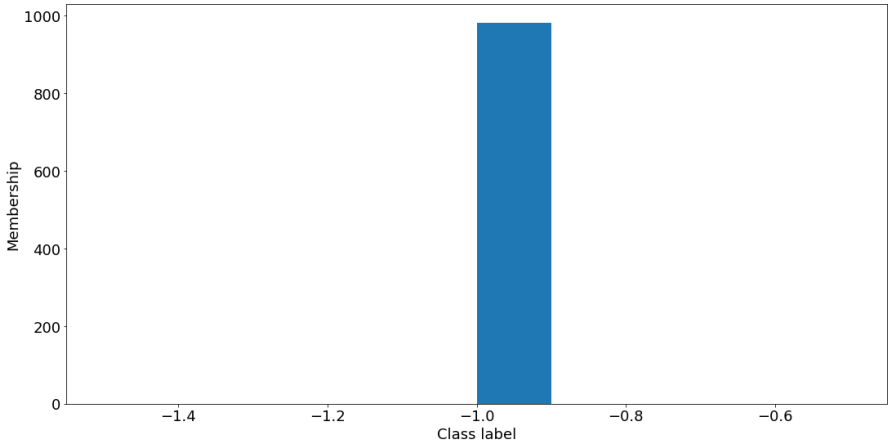

Figure 4.22: Lack of intraclass fragmentation viewed with DBSCAN on bridged MNIST data (unbridged label 4).

that can be taken to change the post-explanation processing pipeline to normalize the data better. It is beneficial to use one *global* set of PCA basis vectors for the entire dataset rather than calculating the basis vectors for each *individual* class, as was done before. Additionally, filtering the inputs to only focus on explanations for instances the network predicted before calculating principal components yields the best results, as this focuses on only the meaningful explanations. Making these changes results in the total class fragmentation graph seen in Figure 4.23. Additionally, because the data is cleaner the variance between classes is more meaningful (visible in Figure 4.24).

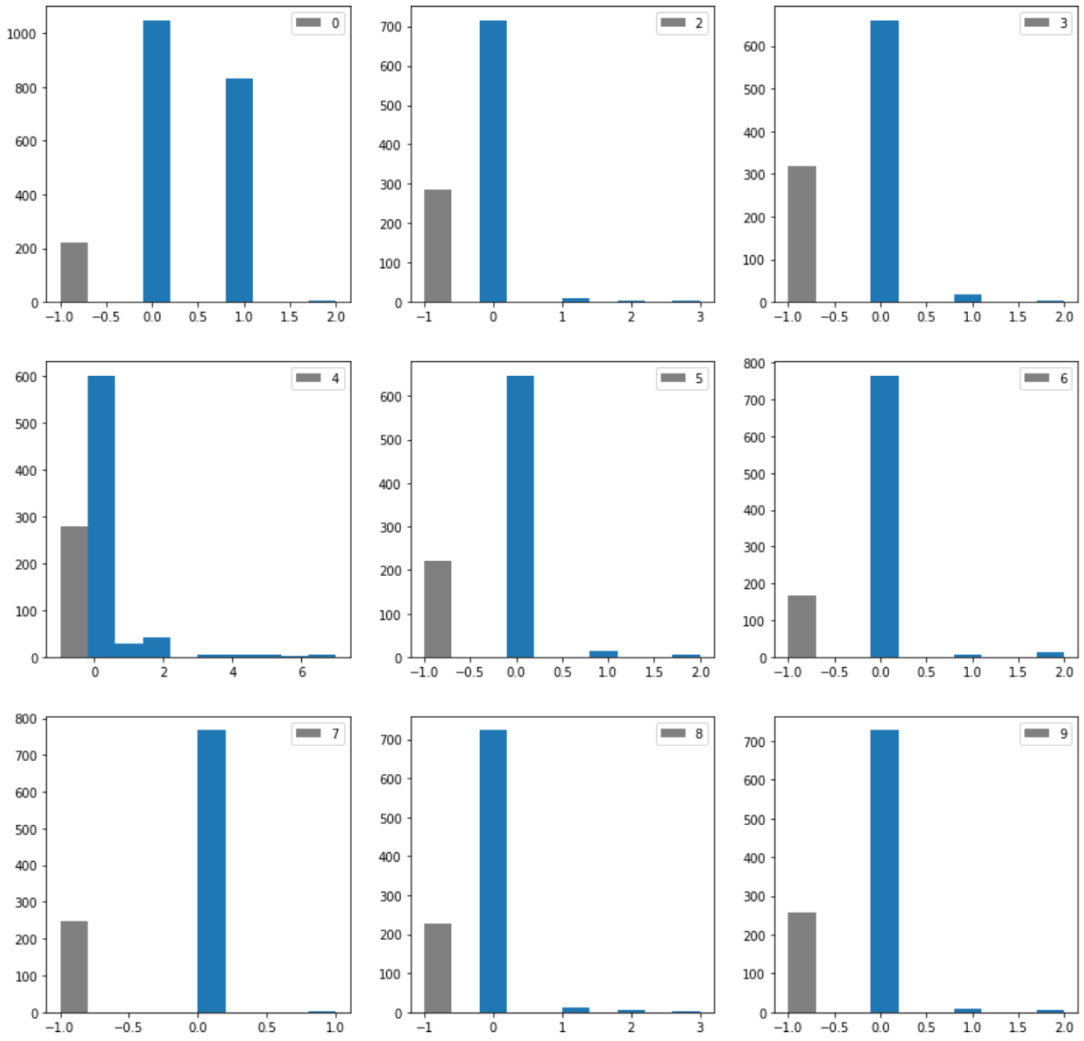

Figure 4.23: Intraclass fragmentation histogram viewed with DBSCAN on bridged MNIST data (bridged labels 0 and 1). As before, the x axis is the class label and the y axis is the count of instances in that label. Note that in the bridged case $0 \to 1$ there are two main bars (+ a gray noise bar) and in the rest there is only one main bar.

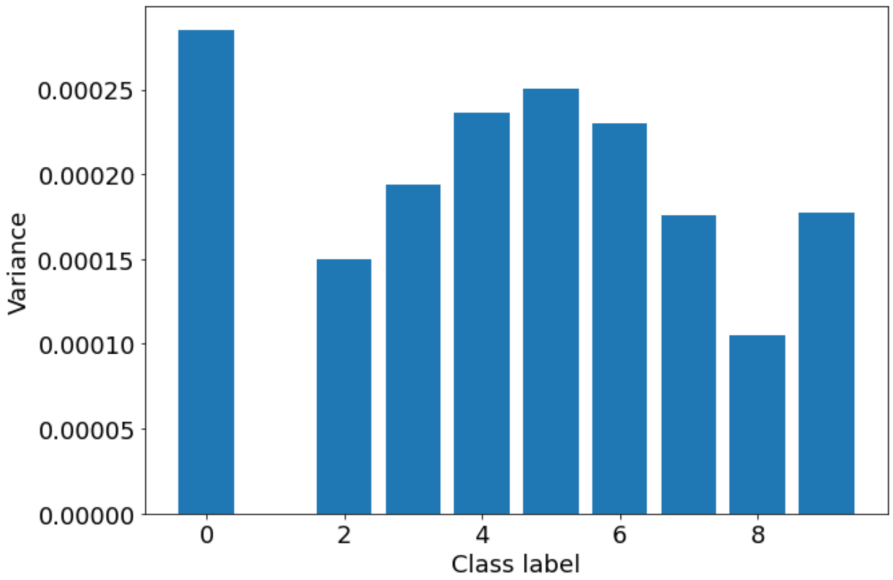

Figure 4.24: Variance inside classes in bridged MNIST data (bridged labels 0 and 1). Note that the variance is highest in class 0 (the bridged class).

Chapter 5

Class Introspection Pipeline

5.1 Setup

5.2 Why a pipeline?

To further test the viability of the class introspection techniques described in Chapter 4, it was necessary to test the method across more data and more instances than just the few MNIST examples in Section 4.3.2. The first step to verifying the results above was to utilize the artificial latent structure technique in Section 4.1 to exhaustively bridge all of the labels in the MNIST dataset in order to compare the performance across all bridged labels.

The pipeline thus needs to be able to process the explanations for the entire exhaustive dataset bridging, and then cluster all the explanations at the end in order to evaluate the method's performance. Thankfully, computing the PCA and DBSCAN pass is very well optimized ($\sim$ 600ms even on slow machines), meaning that the pipeline can focus just on computing the explanations first (the expensive part). Additionally, this allows for further experimentation on different explanation processing techniques.

5.2.1 Pipeline challenges and execution

The ad-hoc method used for the initial explorations (Jupyter notebooks run locally) would not scale well to processing large amounts of data, as generating the explanations is computationally expensive and is a relatively slow operation. Additionally, each run generates a lot of data: SHAP (and other techniques) compute the saliency maps for each available label for each instance, quickly leading to ballooning archive sizes.

To compute the SHAP explanations for all $10,000$ test data instances in the MNIST dataset, each 28×28 pixel grayscale instance is transformed into 10 explanations, each of which must be stored. Even worse, each pixel must be stored as a 64-bit floating point number, as the SHAP explanations are a matrix of floating point saliencies rather than the single-byte grayscale pixels in the original dataset. With some multiplication,

we can find the total storage size required per run:

$$28\ pixels \times 28\ pixels = 784\ pixels$$
$$784\ pixels \times 4\frac{bytes}{pixel} = 3136\ bytes$$
$$3136\ bytes \times 10\frac{explanations}{instance} \times 10000\frac{instances}{run} \times \frac{1}{1024^2} \approx 300\ MiB$$

Each run will require 300 mebibytes of storage for just the raw explanations—which assumes that the explanations are stored with no overhead at all and with no ancillary data! In reality, the storage efficiency is much lower, requiring approximately 550 mebibytes for each run.

The number of runs is an issue as well. To properly test the technique, every combination of labels should be bridged. Using combinatorics, we can find the number of required runs:

$$\binom{10}{2} = \frac{10!}{2!(10-2)!} = 45\ runs$$

To store all 45 runs, we need a total of $45 \times 550\ MiB \approx 25\ GiB$ of storage space.

As an additional challenge, all the compute platforms available at the time were insufficient for executing the full all-cases pipeline. The three options available to run the pipeline were a local laptop (with no CUDA support), DICE's `student.compute`, DICE's GPGPU machines, and a Google Compute Engine trial. These all were insufficient for different reasons.

The local laptop was the first try, but unfortunately each run took about 35 minutes to process—during which time the computer was unusable. Additionally, thermal throttling kicked in, slowing down the process even more. As knocking out a personal laptop for a full day was not an option (thanks to coursework deadlines)—and as to not melt the laptop into a puddle of aluminum—an alternate solution was required.

The School of Informatics' DICE `student.compute` and GPGPU machines were the next try, but unfortunately the disk space requirements made that solution non-viable. The python environment itself is about 6 gibibytes (mainly due to tensorflow), and with only 11 gibibytes allocated to DICE home directories even just installing the environment was a strain the available system resources. 25 gibibytes would simply not fit on DICE. Saving runs outside of the home directory was not trivial, as the request to allocate additional disk space was denied due to funding requirements. Persisting the runs to `/disk/scratch` would not work due to insufficient privileges, and saving to `/tmp` would not work as DICE's temporary directories are a mix of disk and tmpfs (RAM)—not to mention the risk of the runs disappearing before they could be collected. An intermediate solution would be to run the pipeline one case at a time and `scp` them to a personal server with sufficient space, but this too is brittle and significantly slows the processing of the dataset, as between runs processing needs to stop to allow for the explanations to upload.

Key	Type	Description
`pair`	Integer tuple	Pair of labels for the bridging process.
`y_trn_hw`	Numpy array with dtype `float64`	Bridged labels (training)
`y_tst_hw`	Numpy array with dtype `float64`	Bridged labels (test)
`model_name`	String	Name of model in Keras session
`metrics`	Object with keys `loss`, `acc`	Loss and accuracy from the training.
`shaps`	Numpy array with dimensions $(10, 10000, 28, 28)$	SHAP values for each instance on each class. Dimensions correspond to (label, instance, image Y, image X)
`time`	Integer	Time to complete run (seconds)

Table 5.1: Contents of pipeline run output pickle.

The solution that ended up powering the pipeline was Google Cloud Platform (GCP)'s free tier. As an introductory offer, GCP offered a 300$ credit to new accounts to spend on the platform—though with significant limits on the type of machine available to rent. The most unfortunate limitation was that trial accounts could not rent servers with more than eight cores, nor could they rent servers with GPUs. However, these accounts can rent floating block storage devices (of less than 500GiB) and compute machines with less than or equal to eight cores, so that was the solution. The chosen machine was the largest available, the 8 vCPU/32GiB RAM compute-optimized `c2-standard-8`. This instance was able to compute all 45 cases in about 16 hours, and saved them to a floating storage device so the VM could be downsized later to save money.

5.2.2 Pipeline details

The pipeline itself is implemented trivially as a series of steps to produce a Python native pickle file containing a native Python `dict`. Each combination of input label pairs across the dataset is enumerated, yielding a simple list of (A, B) pairs (45 in the MNIST set). For each pair, the corresponding bridged training and test labels were produced from the original MNIST dataset labels.

From these labels, a simple neural network (with the architecture described above in Section 4.3.2) was trained on the training set with the bridged training labels. The final accuracy and loss function values were recorded, and the network was fed to the SHAP explainer with instances in chunks of 25 (inputs chunked simply in order to track execution time). The SHAP explanations were then extracted, and the final outputs were cached as a pickle file onto the GCP floating storage device. See Table 5.1 for a full description of the contents of the pickle.

5.3 Dataset exploration

Of course, just computing the explanations were not useful yet—they need to be processed in a manner conducive to human interpretation. To that end, the next step in

the project was the construction of a simple web app that could be hosted on GCP and interface with the explanations directly.

The web app was built using the Python library `aiohttp`[30]. `aiohttp` provides a web server framework built on top of Python's native asynchronous event loop, `asyncio`[25]. Building on top of the native event loop had several advantages, the main benefit being the ease of interoperability between components of the server as all execution is single threaded and control is explicitly yielded—thereby bypassing a lot of the headaches in traditional threaded or multiprocessed web frameworks. Though `asyncio`-based applications excel at I/O-bound tasks and struggle with CPU-bound tasks, background processes can be fairly simply dispatched with the use of Python's native `multiprocessing`[28] and `concurrent.futures`[26] standard packages.

The web app was implemented as a Javascript Single Page Application (SPA), an application methodology where the web page itself manages all the state and calls into the server's API endpoints to render content (as opposed to server-side rendering, where the page itself is prepared by the server). The web server is in charge of loading the pickled run files from disk and running the post-processing (PCA and DBSCAN from Section 4.3.2) over the specified run. This allows for the DBSCAN epsilon value to be modified on the fly by the end user of the experiment browser.

The frontend was powered by Vue.js[29]. Vue.js allows for the creation of complex web applications without necessitating a build step (e.g. with Webpack or similar), enabling quick setup and prototyping without needing to set up a full toolchain. The frontend was implemented as a single page which rendered the experiments as simply as possible, performing HTTP requests to the server to fetch experimental data as necessary. A screenshot of this application can be seen in Figure 5.1.

The application displays a list of the currently available runs, and when one is clicked on the application runs the PCA and DBSCAN portion of the pipeline with the user-specified epsilon value and serves it to the client along with a set of supplemental information. The client displays all the information in Table 5.1, as well as the total variance explained by the PCA vectors for each class and a histogram of cluster label membership for each class. Using this information, it is trivial to judge how well the pipeline performed over the dataset: if the artificially latent class has a clear split between two labels where other classes do not, then the pipeline was successfully able to detect the latent structure. The exact results over the initial pipeline runs are discussed in Chapter 6.

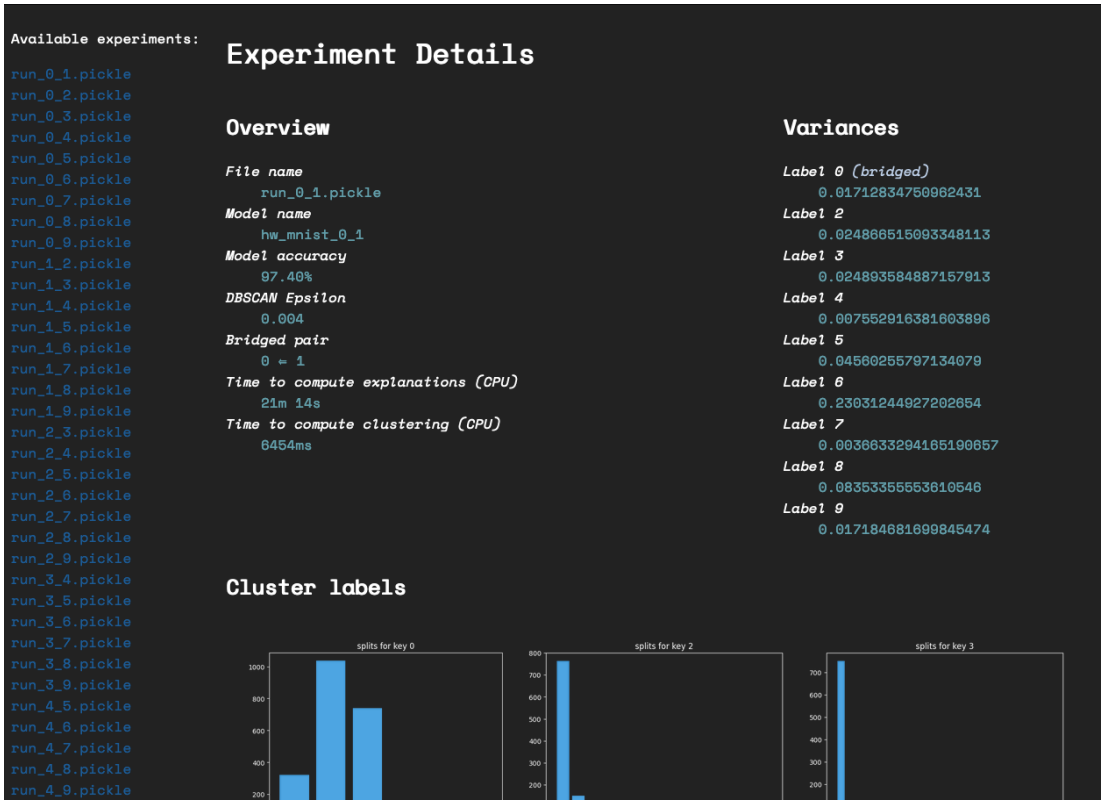

Figure 5.1: Screenshot of the web application interface.

Chapter 6

Discussion

6.1 Comparison to Baseline

Compared to the baseline method described in Section 3.3, the SHAP + PCA + DBSCAN method described above is very effective at determining the latent structure. This variation in performance is indicative of the difference in the type of data being processed. The baseline operates directly over the glyphs themselves, while the class introspection pipeline operates over the classifier's *explanations* of the glyphs. While similar, the distinction is important: in the baseline, the precise positions of the pixels are salient to the class representation, whilst in the explanations it is the specific neuron firings (and their intensities) that are salient. This allows the class introspection pipeline to group positive classifications by the specific neurons that are firing; the intuition being that the exact structure of the glyph does not matter so long as the specific neurons with that class are active which is fine tuned already by the classifier. Compare this to the baseline, which is relegated to determining the specific pixels that make up a class without the benefit of a trained neural network behind it.

6.2 Limitations

This experiment has shown that class introspection is a viable technique, but there are still several limitations in its current iteration. One main issue is that the explainability methods are imperfect. In most cases, SHAP or LIME produce saliencies that are reasonable, but in others they pick up on specific pixels that are assigned a saliency much higher than it's neighbors—or in fact, any other pixel in any other instance—by several orders of magnitude. This is infrequent, but care must be taken to avoid these outliers skewing the PCA vectors. Otherwise, they can completely ruin results, as seen in Figure 6.1.

Another issue with PCA is its ill-suitedness to image data. PCA does not preserve the structure of the image, rather flattening it out into a single 784×1 vector (from 28×28). This flattening ruins any sort of spatial correlation between features, and additionally makes it extremely fragile to image scaling, rotation, transposition, etc.

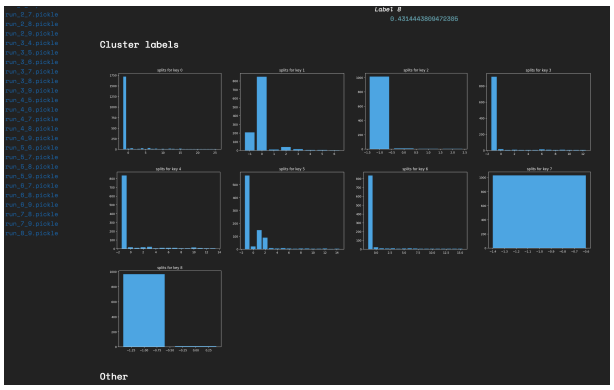

Figure 6.1: Screenshot of the web application interface showing a failed run with the "vanilla" pipeline configuration

A better choice for future work is a proper computer-vision-based feature extraction method, such as Scale-Invariant Feature Transform (SIFT) or Histogram of Oriented Gradients (HOG). These techniques would preserve the feature positions in the saliency map, and allow for a higher-quality dimensionality reduction.

Fundamentally, class introspection is limited by the characteristics (and statistical power) of the classifier being explained. As seen in Section 4.2.3, bridging a low-class-count dataset simply caused the tree-based and multinomial-naïve-Bayes-based methods to ignore details of the bridged dataset—and this can happen with real latent structure as well. If the classifier does not learn the differentiating features of a class with latent subclasses and instead just treats that class as a catch-all, then class introspection is less effective at discovering that subclass. This is why the neural-network-based approaches were effective: the MNIST data was complex enough that the neural network had to actually understand the input features, and this allowed the bridged class to be easily discovered.

More fundamentally still, class introspection is an "unknown-unknown problem": namely, the number of latent classes is unknown—and even worse, the very existence of those latent classes is unknown! It is not unlike searching for a needle in a haystack without even knowing if the needle is there. Additionally, there is another problem: of these discovered subclasses, there is no way of knowing which ones of those subclasses are intentional or which are novel. This means that there will always have to be a human in the loop to determine which subclasses are relevant.

6.3 Future Work

The class introspection pipeline as it stands performs well in this paper, but there are areas that can be improved for more robust performance on a wider variety of data. A simple area of improvement is, as noted above, the dimensionality reduction technique: replacing PCA with a computer-vision-based technique with performance invariant to transformations is an easy way to significantly improve performance. Even operating over a list of keypoints instead of the full image could give very good results, espcially

on complex image data.

Another direction of exploration is the even more involved technique of comparing not only the surface-level saliency but the full neuron activation chain through the network for any particular instance. This would limit the technique to only neural networks, but may be a powerful tool for identifying subclasses. Again, more exploration is needed.

This technique has far reaching possibilities, as the ability to reliably discover latent classes has implications across diverse fields of data science and artificial intelligence research. For example, the ability to reliably detect latent structure in classifiers greatly benefits medical studies who may be looking for a robust alternative to subgroup analysis. More fundamentally, this technique may allow for the auditing of classifiers to ensure that the data they are trained on and the data that they work over is free of unknown and potentially unwanted subclasses that could reduce the overall effectiveness of the classifier.

Even more fundamentally still, this technique could be applied to the symbol grounding problem. Briefly, the symbol grounding problem is the difficulty matching the symbols in a formal system to a categorical representation[4], which could potentially be acheived via this technique (i.e. splitting specific classes until a “ground truth” is found).

6.4 Conclusion

Over the course of this project a technique for reliably extracting unlabeled (latent) subclasses from existing datasets has been developed, leveraging explainability techniques to take advantage of the statistical power of complex models. This provided an advantage over other approaches for capturing latent structure, and was demonstrated over an example dataset with artificially-induced latent structure. Additionally, a pipeline for generating explanations over a dataset was created, with a complementary interactive SPA web application for exploring the clusters created. Finally, the limitations of the current iteration of the technique were explored and future areas of work and future applications were expanded upon.

Chapter 7

Bibliography

[1] Edgar Anderson. “The Species Problem in Iris”. In: *Annals of the Missouri Botanical Garden* 23.3 (1936), pp. 457–509. ISSN: 0026-6493. DOI: 10.2307/2394164.

[2] R. A. Fisher. “The Use of Multiple Measurements in Taxonomic Problems”. en. In: *Annals of Eugenics* 7.2 (1936), pp. 179–188. ISSN: 2050-1439. DOI: 10.1111/j.1469-1809.1936.tb02137.x.

[3] R. B. Marimont and M. B. Shapiro. “Nearest Neighbour Searches and the Curse of Dimensionality”. In: *IMA Journal of Applied Mathematics* 24.1 (Aug. 1979), pp. 59–70. ISSN: 0272-4960. DOI: 10.1093/imamat/24.1.59.

[4] Stevan Harnad. *The Symbol Grounding Problem*. http://cogprints.org/3106/index.html. Journal (Paginated). 1990.

[5] Martin Ester et al. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: AAAI Press, 1996, pp. 226–231.

[6] D. Gordon E. Robertson. *English: Northern Blue Flag (Iris Versicolor), Ottawa, Ontario*. June 2005.

[7] Yann LeCun and Corinna Cortes. “MNIST Handwritten Digit Database”. In: (2010).

[8] Julian PT Higgins and Sally Green. *Cochrane Handbook for Systematic Reviews of Interventions*. https://handbook-5-1.cochrane.org/chapter_9/9_6_2_what_are_subgroup_analyses.htm. Mar. 2011.

[9] Ian Jolliffe. “Principal Component Analysis”. en. In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer, 2011, pp. 1094–1096. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_455.

[10] Bruno Scarpa. “Data Mining”. en. In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer, 2011, pp. 336–339. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_455.

[11] Wilfried Siedel. “Mixture Models”. en. In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer, 2011,

pp. 827–829. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_455.

[12] Stephanie T. Lanza and Brittany L. Rhoades. "Latent Class Analysis: An Alternative Perspective on Subgroup Analysis in Prevention and Treatment". In: *Prevention science : the official journal of the Society for Prevention Research* 14.2 (Apr. 2013), pp. 157–168. ISSN: 1389-4986. DOI: 10.1007/s11121-011-0201-1.

[13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: *arXiv:1312.6034 [cs]* (Apr. 2014). arXiv: 1312.6034 [cs].

[14] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *arXiv:1602.04938 [cs, stat]* (Aug. 2016). arXiv: 1602.04938 [cs, stat].

[15] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 4765–4774.

[16] F. K. Došilović, M. Brčić, and N. Hlupić. "Explainable Artificial Intelligence: A Survey". In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 2018, pp. 0210–0215. DOI: 10.23919/MIPRO.2018.8400040.

[17] Stefano Teso and Kristian Kersting. ""Why Should I Trust Interactive Learners?" Explaining Interactive Queries of Classifiers to Users". en. In: *arXiv:1805.08578 [cs, stat]* (May 2018). arXiv: 1805.08578 [cs, stat].

[18] Colin Paterson and Radu Calinescu. "Detection and Mitigation of Rare Subclasses in Neural Network Classifiers". en. In: *arXiv:1911.12780 [cs, stat]* (Nov. 2019). arXiv: 1911.12780 [cs, stat].

[19] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. "Learning Important Features Through Propagating Activation Differences". In: *arXiv:1704.02685 [cs]* (Oct. 2019). arXiv: 1704.02685 [cs].

[20] Stefano Teso. "Toward Faithful Explanatory Active Learning with Self-Explainable Neural Nets". en. In: (2019), p. 13.

[21] Bhavya Ghai et al. "Explainable Active Learning (XAL): An Empirical Study of How Local Explanations Impact Annotator Experience". In: *arXiv:2001.09219 [cs]* (Sept. 2020). arXiv: 2001.09219 [cs].

[22] Iain Murray and Arno Onken. *Autoencoders and Principal Components Analysis (PCA)*. en. https://mlpr.inf.ed.ac.uk/2020/notes/w9a_autoencoders_pca.html. Lecture. 2020.

[23] Ali Abdalla. "A Visual History of Interpretation for Image Recognition". In: *The Gradient* (2021).

[24] Peter Bell. "Automatic Speech Recognition Course Notes". In: (2021).

[25] Python Software Foundation. *Asyncio — Asynchronous I/O*. https://docs.python.org/3/library/asyncio.html. Mar. 2021.

[26] Python Software Foundation. *Concurrent.Futures — Launching Parallel Tasks*. https://docs.python.org/3/library/concurrent.futures.html. Mar. 2021.

[27] Scott Lundberg. *Slundberg/Shap*. Mar. 2021.

[28] Python Software Foundation. *Multiprocessing — Process-Based Parallelism*. https://docs.python.org/3/library/multiprocessing.html. Mar. 2021.
[29] Evan You. *Vue.Js*. en. https://vuejs.org/. 2021.
[30] aiohttp maintainers. *Welcome to AIOHTTP — Aiohttp 3.7.4.Post0 Documentation*. https://docs.aiohttp.org/en/stable/.

Appendix A

Supporting Code

This code for this project is available on Github, at the link below:

`https://github.com/pkage/class_introspection`

The project's `README.md` contains the layout of the code, as well as instructions for creating the environment and running the specific Jupyter notebooks containing the experiments.