SUPPLEMENTARY MATERIAL

A DERIVATIONS

Below, we prove Proposition 1, which states

Proposition 1. *For a classification model f,* $\arg\min_{x'} \ell(f, x', y') \in \arg\min_{x'} h(x')$*, where $l(\cdot)$ is cross-entropy and $h(\cdot)$ is predictive entropy.*

We start by introducing the notation, definitions, and our assumptions.

Let $p_f(y'|x')$ denote the softmax probability assigned to class y' for input x' by the classifier f. Let $\mathcal{X}$ and $\mathcal{Y}$ denote the domains of x and y, respectively. We assume $0 \cdot \log(0) := 0$; $\log(0) := -\infty$; and $p_f(y|x) > 0, \forall x \in \mathcal{X}, y \in \mathcal{Y}$.

For simplicity, we provide the derivation for a single input $x' \in \mathcal{X}$ with target class y'. However, the proof can be easily extended to multiple observations. When generating a CE x' targeted to class y' we minimize the cross-entropy loss, defined as

$$\ell(x', y') = -\log p_f(y'|x'). \tag{13}$$

We observe that the function obtains a minimum at $p_f(y'|x) = 1$ and $p_f(y|x) = 0 \quad \forall y \in Y \backslash y'$ for which $\ell(x', y') = 0$. This is a unique minimum because [1] cross-entropy is bounded below by 0 and [2] it is monotonically decreasing in $p_f(\cdot)$.

Predictive entropy, $h(x')$, is defined as

$$h(x') = -\sum_{y \in \mathcal{Y}} p_f(y|x') \log p_f(y|x'). \tag{14}$$

If $p_f(y'|x) = 1,$ and $p_f(y|x) = 0 \quad \forall y \in Y \backslash y'$, then $h(x') = 0$. This is a minimum because predictive entropy is also bounded below by 0.

B COUNTERFACTUAL EXPLANATION GENERATION ALGORITHM

Algorithm 2 Generating Counterfactuals

Input original observation x; target class y'; ensemble of models $\{f_m\}_{m=1}^{M}$; maximum number of iterations N; minimum confidence of target class γ; perturbation size δ; maximum number of times each feature is changed n; optional: a function that clips the values to a permitted pre-defined range clip.
Output counterfactual x'
$x' \leftarrow x$
$c \leftarrow 0$
Create a n_p vector, where n_p, the number of input features, with all values initialized to zero, $P = 0_{n_p}$ This vector will ensure keep track of the number of times each feature has been changed.
while $\bar{p}(y'|x') < \gamma$ and $c < N$ **do**
    Compute forward derivative: $S(x', y') = \nabla_{x'} \frac{1}{M} \ell(f_m, x', y')$
    Select the most salient feature: $i = \text{argmax}_{i:i \in P, P[i]<n} S(x', y')[i]$
    Update the most salient feature by δ: $x'[i] = x'[i] + \text{sign}(S(x', y')[i]) \cdot \delta$
    Clip the counterfactual so that it stays within the pre-defined range: $x' = \text{clip}(x')$
    $P[i] \leftarrow P[i] + 1$
    $c \leftarrow c + 1$
end while
return x'

C ADVERSARIAL TRAINING

Recent work in adversarial literature has linked adversarial robustness to improving model interpretability (Tsipras et al., 2019). Improving adversarial robustness can be achieved through adversarial training, corresponding to minimizing the loss:

$$\min_{\theta} \mathop{\mathbb{E}}_{p_\theta(y|x)} \left[\max_{\delta \in \Delta} \ell(x + \delta, y|\theta) \right], \tag{15}$$

where θ are the model parameters, ℓ is the loss function, x is the original input, y is the original class, δ is a perturbation, and Δ is set of possible perturbations.

In practice, adversarial training is often implemented by augmenting the dataset with adversarial examples during training. These additional images ensure that the model does not focus on noise when learning features for classification. Thus, the model is more likely to learn features that are not noise, and therefore are more interpretable (Tsipras et al., 2019; Ilyas et al., 2019). This means that adversarial training can also be used to improve the performance of models, outside of the adversarial literature setting.

Further, Chalasani et al. (2020) show a connection between feature-attribution explanations and adversarial training, finding that it leads to more sparse and stable explanations from [1] an empirical perspective for image data and [2] a theoretical perspective for single-layer models.

Lastly, adversarial training can be used to improve uncertainty estimation. Lakshminarayanan et al. (2017) show that adversarial training is a computationally efficient solution for smoothing the predictive distribution, and can improve the accuracy and calibration of classifiers in practice.

The aforementioned work motivates the use of adversarial training to generate more interpretable, stable explanations.

D EVALUATION

We omit IM2 evaluation as we found that it failed to pass a sanity check that compares the metric for in- and out-of- distribution images. We perform the check using MNIST, as it made it easier to visually verify test results.

We use MNIST as in-distribution data, and EMNIST as out-of-distribution data. This means that the autoencoders are trained on MNIST images. For the 'in-distribution' CEs, we take MNIST training data in the target class – these images can be considered as 'gold standard' CEs. For the 'out-of-distribution' CEs, we select a random

image from EMNIST. We expect to see a clear different in the IM2 scores for the in- and out-of-distribution data as we are comparing the gold standard CEs with random images from a different dataset.

However, the right-hand plot of Figure 5 demonstrates that the IM2 scores for out-of-distribution did not differ significantly than IM2 scores of in-distribution data at a 5% significance level. On the contrary, for IM1, we find a significant difference (measured using a t-test) for in- and out-of distribution IM1 score at a 5% significance level. Visually, the difference can be observed in the left box-plot in Figure 5.

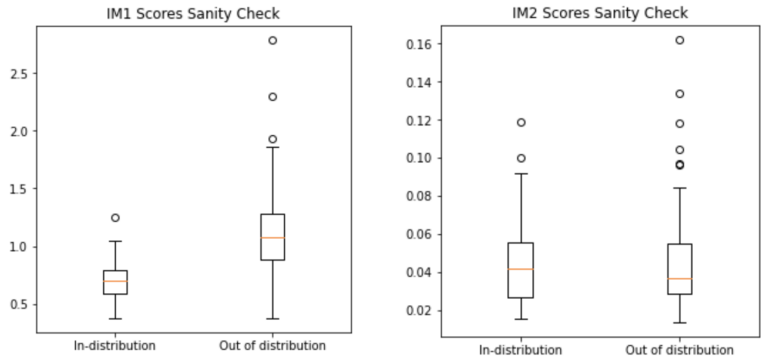


Figure 5: IM1 and IM2 scores for in-distribution data versus out-of-distribution data

E EXPERIMENT SETUP

Below, we summarize the experiment setup for the different experiments. All experiments are implemented in PyTorch. The code repository contains the instructions for reproducing each result and generating example CEs, alongside the details of the environment setup including the versions of dependencies.

E.1 MNIST

Dataset Configuration For our method and JSMA we normalize the inputs to $[0, 1]$. For Van Looveren and Klaise (2019) we normalize to $[-0.5, 0.5]$, as suggested. We train the classifier on the training set, and generate CEs on the test set. We tune the hyperparameters of our method by generating CEs on the training set. We did not tune the hyperparameters of Van Looveren and Klaise (2019), as we could select them from the original paper.

For each point in the test set, we select the target class for the explanation by randomly selecting from a set of classes specific to the class of the input image. This allows us to avoid selecting target classes which are difficult for a particular input class, for example transforming a 6 into a 2. For a complete list of the possible target classes for each input class, see the code release.

Model Architecture We use a three-layer MLP with 200, 200 and 10 nodes per layer, respectively. The first two layers have ReLU activations and batch normalization after the activation. The final layer has a softmax activation. We use an ensemble of 50 models.

Optimisation We train the network using mini-batches of size 128 and Adam (Kingma and Ba, 2015) (with default PyTorch hyper-parameters) to optimize the weights. We train the model for 50 epochs.

Our Method: adversarial training We implement adversarial training as follows: For each iteration, perform a training step a (clean) batch of data (X, y) followed by a training step on an adversarially-augmented batch of data. To create the latter, we use FGSM (Goodfellow et al., 2015) to generate adversarial examples X'. The augmented dataset is $(X, y) \cup (X', y)$. We choose $\varepsilon = 0.15$ as the perturbation is large enough to fool a trained classifier 90% of the time (and therefore will improve robustness), however does not change the true classification.

Our Method: hyperparameters The maximum number of permitted changes is 5 and the maximum number of iterations is 3,920. The confidence level, γ, is set to 0.99. The perturbation (δ in the pseudo-code) is set to 0.2 (which is equal to $1/n$, where n is the number of times each feature can be changed).

IM1 Evaluation We implement IM1 evaluation following Van Looveren and Klaise (2019), and include the configuration here for completeness. Table 2 shows the architecture of the all class and single class autoencoders. We train both autoencoders using the mean-squared objective and Adam with a batch size of 128. For the all class autoencoder we train for 4 epochs, for the single class autoencoder we train for 30 epochs.

Table 2: IM1 MNIST autoencoder architectures. Top-down in the table corresponds to input-output in the architecture. The hyperparameters are output channels c, kernel size k, padding p, stride s, scale sc.

All Class		**Single Class**	
Layer	Parameters	Layer	Parameters
Encoder		*Encoder*	
2D Convolution	$c = 16, k = 3, p = 1$	2D Convolution	$c = 16, k = 3, p = 1$
ReLU activation		ReLU activation	
2D Convolution	$c = 16, k = 3, p = 1$	2D Convolution	$c = 16, k = 3, p = 1$
ReLU activation		ReLU activation	
Max-pool 2D	$k = 2, s = 2$	Max-pool 2D	$k = 2, s = 2$
2D Convolution	$c = 1, k = 3, p = 1$	2D Convolution	$c = 1, k = 3, p = 1$
Decoder		*Decoder*	
2D Convolution	$c = 16, k = 3, p = 1$	2D Convolution	$c = 16, k = 3, p = 1$
ReLU activation		ReLU activation	
Upsample	$sc = 2$, mode = "nearest"	Upsample	$sc = 2$, mode = "nearest"
2D Convolution	$c = 16, k = 3, p = 1$	2D Convolution	$c = 16, k = 3, p = 1$
ReLU activation		ReLU activation	
2D Convolution	$c = 1, k = 3, p = 1$	2D Convolution	$c = 1, k = 3, p = 1$

Results aggregation The results in Table 1 are aggregated as follows:

- For each method and dataset pair we choose 100 points from the test set.
- For 10 random seeds
  - Generate a CE for each of the 100 test points
  - Compute the mean IM1 score and L_1 distance
- Compute the mean of the means, and the standard deviation of the means.

Configuration of Van Looveren and Klaise (2019) We use the implementation released by the authors in the ALIBI library (Klaise et al., 2020), largely in its default configuration as given in the documentation. We use the same classifier architecture as above. As the classifier is a white-box model, we follow the ALIBI documentation and use loss function D from Van Looveren and Klaise (2019). Thus, the hyperparameter configuration is $c = 1$, $\kappa = 0$, $\beta = 0.1$, $\gamma = 100$, $\theta = 100$, max iterations $= 2000$. As in Van Looveren and Klaise (2019) we use the encoder to find the class prototypes, and following the ALIBI documentation set $K =$ all instances.

E.2 Wisconsin Breast Cancer Dataset and Boston Housing Dataset

We generally use the same setup for both the Wisconsin Cancer and the Boston datasets, except where we clearly indicate differences below.

Dataset Configuration For our method and JSMA we normalize the inputs to $[0, 1]$. For Van Looveren and Klaise (2019) we standardize the inputs to mean 0 and standard deviation 1, as suggested. We randomly select 70% of each dataset as the train set, 10% as the validation set, and 20% as the test set. We train the classifier on the training set, tune hyperparameters on the validation set, and generate CEs on the test set.

Model Architecture We use a three-layer MLP with 80, 80 and 2 nodes per layer, respectively. The first two layers have ReLU activations and batch normalization after the activation. The final layer has a softmax activation. We use an ensemble of 20 models.

Optimisation We follow the same setup as for MNIST.

Our Method: adversarial training We perform adversarial training similarly to MNIST. Contrary to MNIST, each feature has a different scale and distribution. A list of perturbation sizes can be found in the code in the configuration file in demo_data, after the word perturbation.

Our Method: hyperparameters The maximum number of permitted changes is 5 and the maximum number of iterations is 150. The confidence level, γ, is set to 0.99. The perturbation sizes (δ in the pseudo-code) are feature-specific and can be found in the file breast_cancer_config.txt in the code.

IM1 Evaluation For the Wisconsin Breast Cancer dataset we follow Van Looveren and Klaise (2019), and include the configuration here for completeness. We use the same procedure for the Boston Housing Dataset. We use the same autoencoder architecture for both the all class and single class autoencoders, and Table 3 shows the architecture. We train both autoencoders using the mean-squared objective and Adam with a batch size of 128 for 500 epochs.

Table 3: IM1 VAE architecture for the Wisconsin Breast Cancer and Boston Housing datasets. The top of the table corresponds to the input of the model, and the bottom the output. The hyperparameter n is the number of hidden units.

Layer	Parameters
Encoder	
Linear	$n = 20$
ReLU activation	
Linear	$n = 10$
ReLU activation	
Linear	$n = 6$
Decoder	
Linear	$n = 6$
ReLU activation	
Linear	$n = 10$
ReLU activation	
Linear	$n = 20$

Results aggregation The same as for MNIST.

Configuration of Van Looveren and Klaise (2019) We use the implementation released by the authors in the ALIBI library (Klaise et al., 2020), largely in its default configuration as given in the documentation. We use the same classifier architecture as above. As the classifier is a white-box model, we follow the ALIBI documentation and use loss function B from Van Looveren and Klaise (2019) for both datasets. For both datasets we use the kd-tree approach to find the prototypes. For the Wisconsin Breast Cancer dataset the hyperparameter configuration is $c = 1$, $\kappa = 0$, $\beta = 0.1$, $\gamma = 0$, $\theta = 100$, $k = 1$, max iterations $= 2000$. We set θ by examining the grid search shown in Van Looveren and Klaise (2019, Figure 7). We set $k = 1$ as this is the default value in ALIBI. For the Boston Housing dataset we use the same configuration as for the Wisconsin Breast Cancer dataset. The exception are k and θ for which we perform a grid search similar to that run by Van Looveren and Klaise (2019, Appendix A) for the Breast Cancer dataset, based on which we choose $k = 10$ and $\theta = 100$.

We note that the IM1 scores we report for Van Looveren and Klaise (2019) are worse than those in their paper. They report an IM1 score which is not significantly different to ours. However, their results are not directly comparable because we use a test set of 100 points while the original paper uses one of 19 points. To try and ensure our results are correct, we took the following steps:

- Used the implementation of the generation algorithm provided by the authors
- Ensured inputs to the generation algorithm are standardized as in the original paper

- Ensured we use the same hyperparameters as in the original paper
- Visually examined the CEs generated by the method to ensure they were reasonable

We repeated the experiment using a test set of 19 points rather than 100, but this also did not reproduce the results in the original paper.

F ADDITIONAL FIGURES

F.1 Uncertainty And Calibration

F.1.1 Calibration

Calibration is important as we assume that our classifier outputs $p(y|x)$. We investigate the effect of the number of components in the ensemble by performing a similar experiment to (Lakshminarayanan et al., 2017, Figure 6), and considering accuracy as a function of confidence. We train our network (same configuration as in Appendix E) on MNIST. We test the network on a dataset formed by combining MNIST and FashionMNIST, where we increase the proportion of the dataset taken from FashionMNIST until the confidence of the classifier falls to the desired level. Ideally the network will have a low-confidence for incorrect predictions. Figure 6 shows the performance of a single classifer, and ensembles with between 5 and 50 components. We can see that the ensembles perform better than a single classifier as their accuracy is higher.

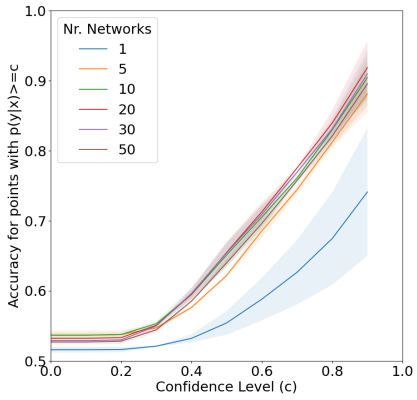
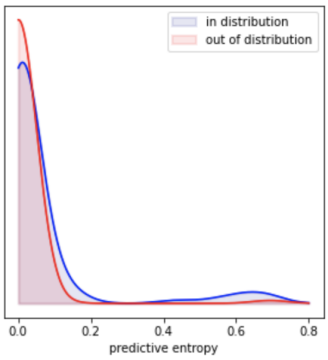


Figure 6: Accuracy of the model as a function of confidence, for ensembles containing between 1 and 50 models. The solid line shows the mean, and the shaded area shows the 95% confidence level over 10 random seeds. A higher line is better.

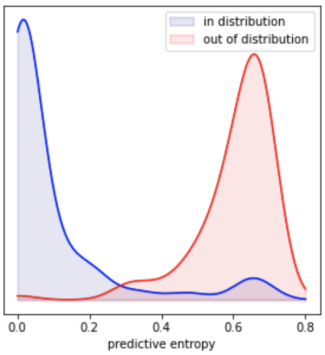
F.1.2 Out-of-Distribution Uncertainty

Our method is reliant on the quality of the uncertainty estimates offered by the classifier, thus we investigate the effect of ensembling and adversarial training on the uncertainty estimates. We measure epistemic and aleatoric uncertainty using predictive entropy. Ideally, we observe a high predictive uncertainty for out-of-distribution data and a low predictive uncertainty for inputs similar to the training data.

We consider in-distribution to be test samples from the Breast Cancer Dataset, and out-of-distribution is tabular data sampled from a normal distribution. Figure 7 shows the effect of using adversarial training and ensembling on predictive uncertainty. While a single model cannot distinguish between in- and out-of-distribution inputs, when using ensembling and adversarial training we see a clear separation of the predictive entropy of both sets of inputs. This goes some way to explain why we see improved performance in the ablation study as we increase the number of models in the ensemble and include adversarial training.



No Adversarial Training or Ensembling



Adversarial Training and Ensembling

Figure 7: Predictive entropy of the model for in- and out-of distribution inputs. The left plot shows a single model without adversarial training or ensembling. The right plot shows an ensemble of 50 models with adversarial training.

F.2 Qualitative Analysis

Figure 8 shows more qualitative examples of counterfactuals generated by our algorithm on MNIST. The first column shows the original class; the second column shows the counterfactual; the third column shows the proposed change. In the third image in each tuple, black denotes *"painting the pixels black in the original image"*, white denotes *"painting the pixels white in the original image"* and gray denotes *"no change"*. For example, consider the first row in Figure 8. The goal is to change a 0 into 3. Our model proposes:

- adding an additional stroke in the middle (shown by the white pixels in the third image),
- removing some pixels so that the from the left and right part of the 0 (shown by the black pixels in the their images).

Both changes are required to create a realistic 3. We observe that, in general, our model can grasp high-level changes that are required. These changes suffice for explanatory purposes. However, our model does not capture stylistic properties, which be seen from the examples in the third row, left side in Figure 8. Further work is required if we want to employ our method as a generative model. An example of a 'failure case' is on left side of the last row – this example is particularly challenging for our model. However, on a high-level we can see that the algorithm roughly understands the required changes.

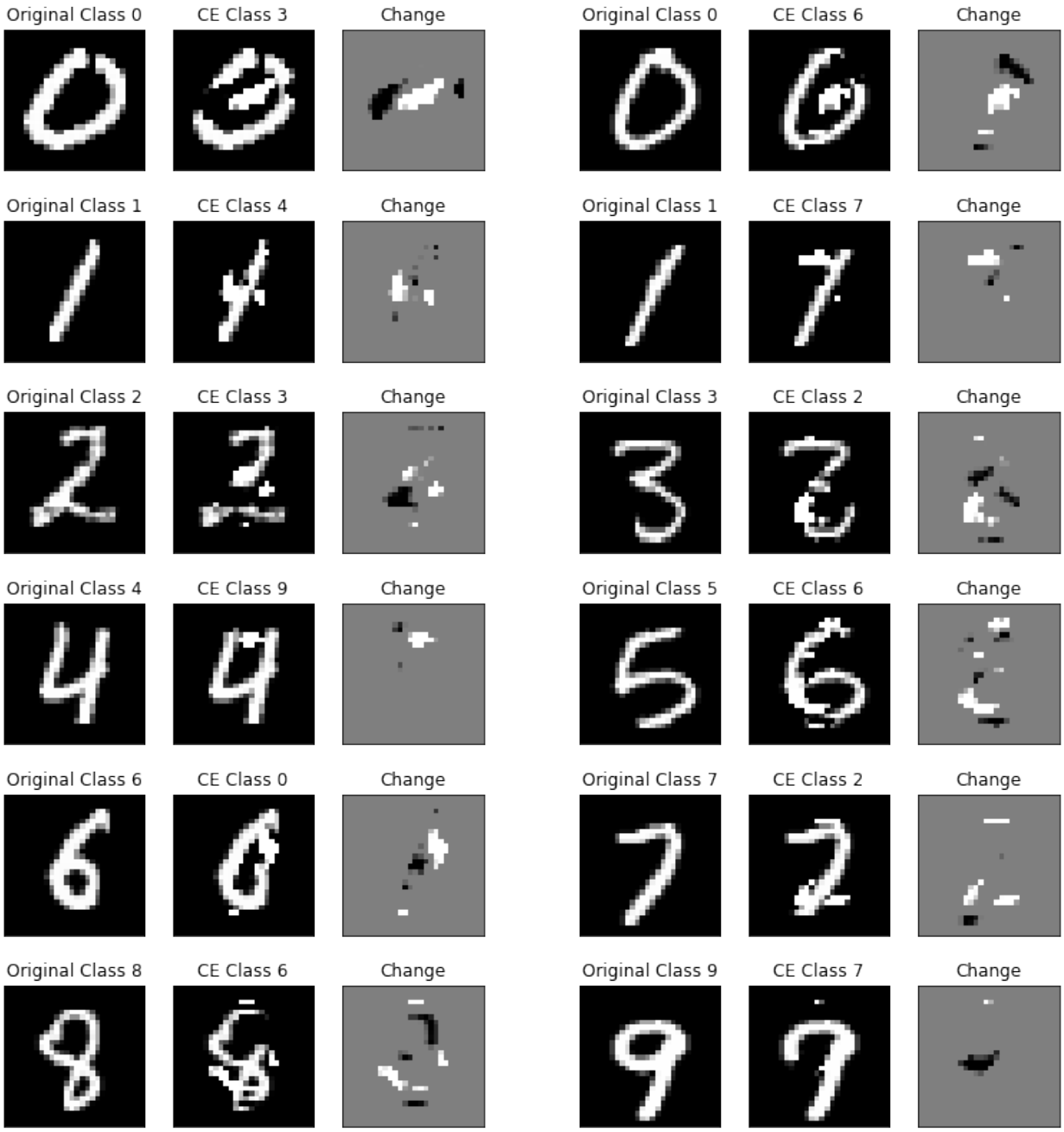


Figure 8: Qualitative Examples of Generated CE on MNIST. The first column shows the original image, the second column the counterfactual, and the third column the proposed change (to the original image to generate the counterfactual).

Figure 9 shows more qualitative examples of counterfactuals generated by our algorithm on FashionMNIST. Again, we observe that our model is able to grasp high-level changes required, such as adding a split to the dress or a zipper to the coat.

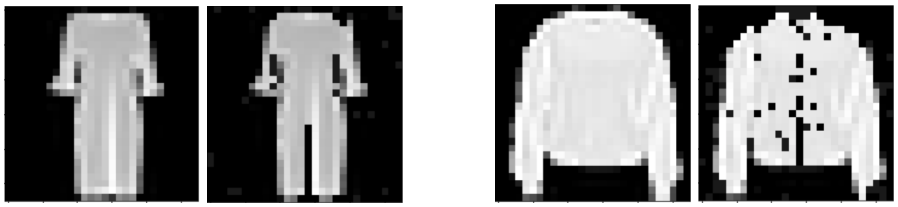


Figure 9: Qualitative Examples of Generated Counterfactuals for FashionMNIST. For each pair, the left image shows the original images: a dress (left pair) and pullover (right). The right images below show generated counterfactuals: pants (left pair) and a coat (right pair).

F.3 Feature Analysis

The importance of a feature j can be roughly estimated by $\sum_i x_{i,j} - x'_{i,j}$, where i denotes the observation. Using this, we determine which features are most frequently changed. In Figure 11, we show examples for MNIST. Here features denote pixels that are changed.

For each triple, the left image shows the average input image from the training dataset – this represents a prototype of the MNIST digit. The blurriness is caused by the natural variation of the class within the dataset. The second image in each triplet is the average CE generated for the target class shown above the image. The third row shows the most average pixel change, i.e. $1/n \sum_i x_{i,j} - x'_{i,j}$. In the third image, black denotes *"painting the pixels black in the original image"*, white denotes *"painting the pixels white in the original image"* and gray denotes *"no change"*.

Overall, the proposed changes appear to be realistic. Let consider a specific example: a counterfactual explanation that changes the predicted class from 0 to 6, as shown in the first row of Figure 11. In Figure 10, we highlight the key changes that can be read from Figure 11. The red circles show the parts that have been *removed* from the original input (left image) to create the counterfactual (shown in the middle image). To change a zero into a 6, we need to paint some pixels black at the top right – these pixels are shown in black in the average change plot (i.e., right image). The green circles show the parts that have been *added* to the original input (left image) to create the counterfactual (shown in the middle image). To change a zero into a 6, we need to add a white diagonal stroke – these pixels are shown in white in the average change plot (i.e., right image).

Figure 10: Average Feature Perturbation for MNIST. In each tuple: the left image shows the average original input image; the middle image is the average CE; the right image is the average change.

Figure 12 shows the most frequently changed cell nuclei properties, and Figure 13 shoes the average perturbation size per changed feature. Further interpretation of these results requires expert knowledge, which we intend to look into for future work.

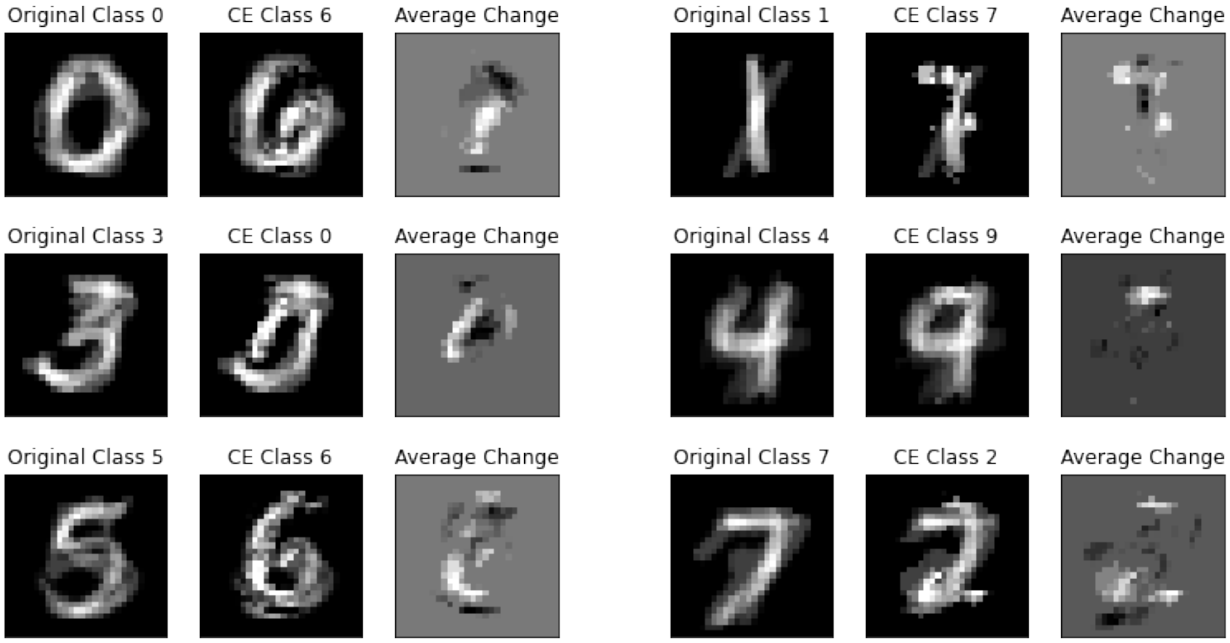


Figure 11: Average Feature Perturbation for MNIST. The left image shows the average original input image. The middle image is the average CE. The right image is the average change.

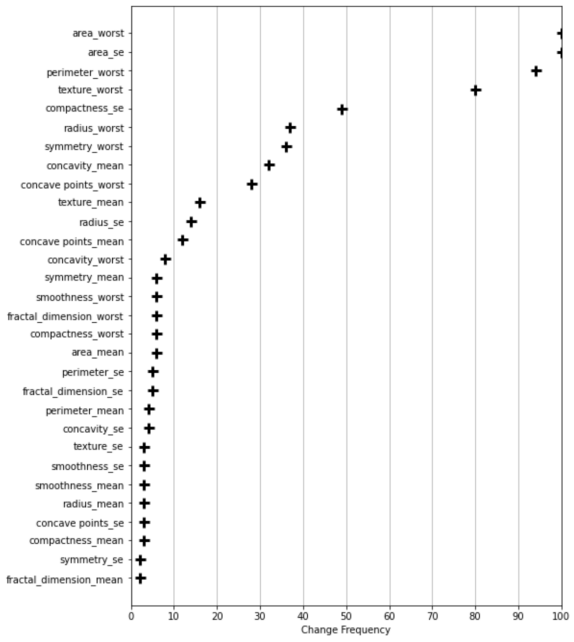


Figure 12: Frequency at which features are changed in counterfactual explanations

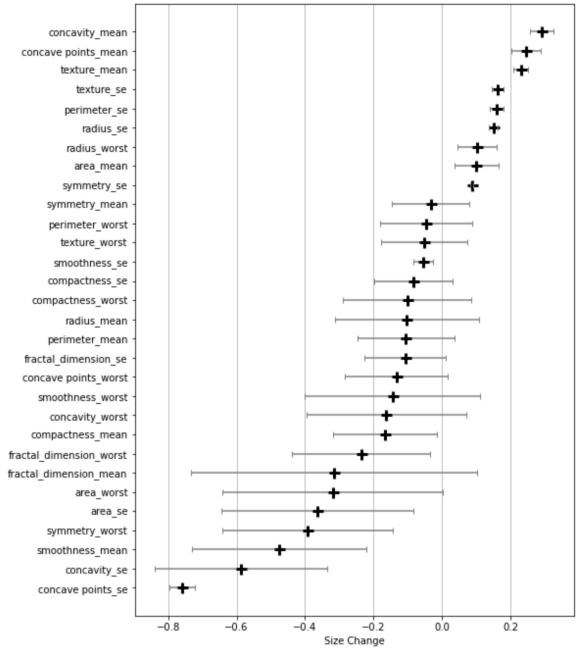


Figure 13: Average Feature Perturbation (of altered features)