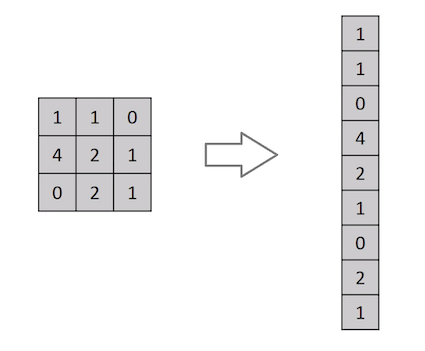
**Internship Summary**

In this internship I needed to use convolutional neural networks to extract features from user submitted pictures. The pictures are submitted for facial recognition when users are applying for a loan. By analyzing the relationship between extracted features and loan overdue risk, I selected important features and added them to the original risk model. I then compared the performance of the new risk model to the original risk model. To do this I had to learn about convolutional neural networks and pytorch.

**CNN Introduction**

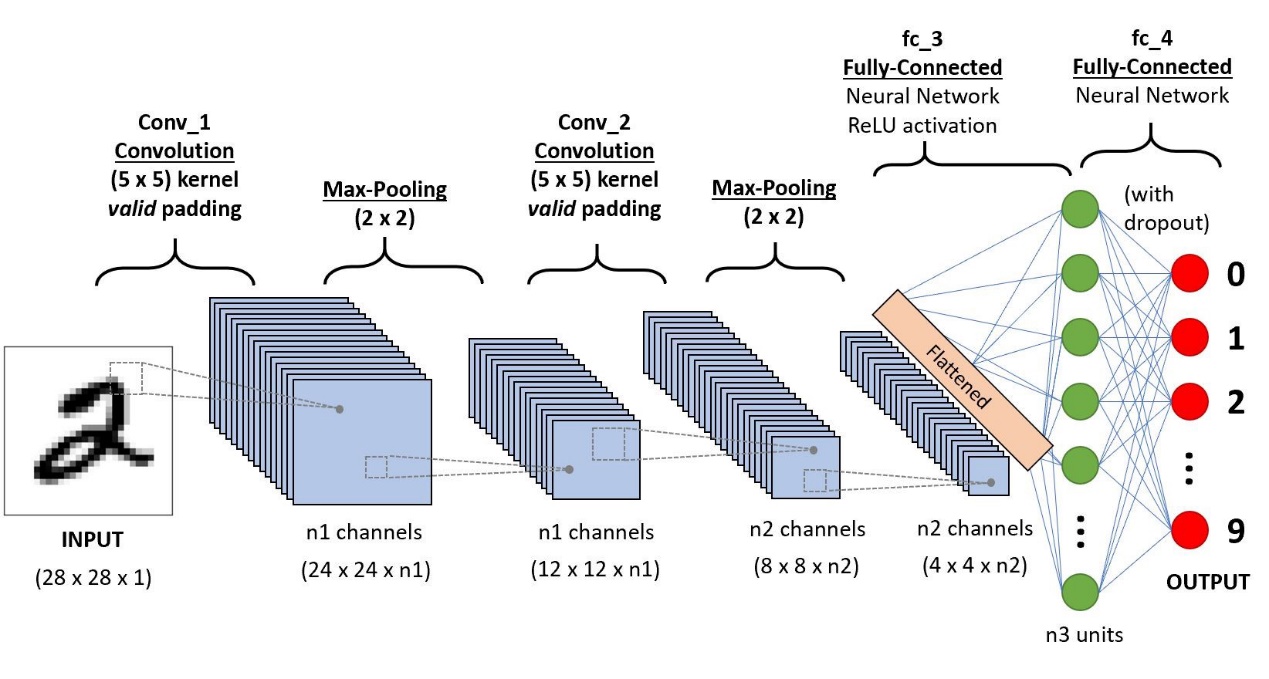
A convolutional neural network (CNN) is a deep learning algorithm used for classification tasks. CNNs can take an input of an image and output the probabilities of different features in the image. The structure of CNNs was inspired by the structure of neurons in the human brain.

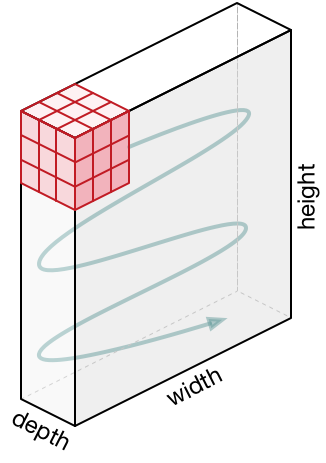


CNNs are more effective than simple fully connected deep neural networks at image classification tasks. This is because in the process of flattening the input data for a fully connected deep neural network, spatial information is lost, while CNNs can retain spatial information and recognize patterns in the image data. CNNs also have a far lower number of parameters than fully connected neural nets, making them easier to train and run.

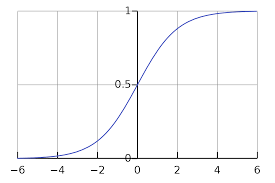
**CNN structure**

CNNs are made from multiple convolution layers which eventually feed into some fully connected layer once higher-level information has been extracted.





CNNs utilize a kernel which moves across the original image and applies a filter. The amount the kernel moves each time is called the stride. At every step the kernel does a convolution operation with the input data, using learned weights, to output new features. Multiple kernels with different weights are used on the input to create multiple new convolved features (output channels), as shown above.

CNNs usually have more than one layer. The first few convolution layers extract lower level features such as edges and colour. With more layers, CNNs can also extract higher level features using the same architecture.

After going through convolution, the data is then non-linearized using the ReLU function. Putting the data through a non-linear function is important because most real-life patterns cannot be approximated using just a linear function. The ReLU function simply returns 0 when the inputed value is negative. ReLU is used over other functions for non-linearization such as a sigmoid function. Sigmoid non-linearization can cause gradient vanishing issues in neural networks due to gradient saturation. It is also easier to calculate derivatives on a ReLU function for backpropagation.

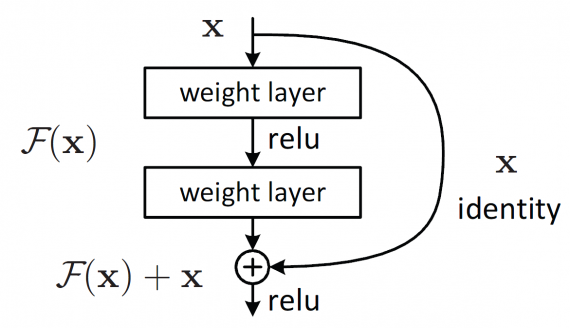


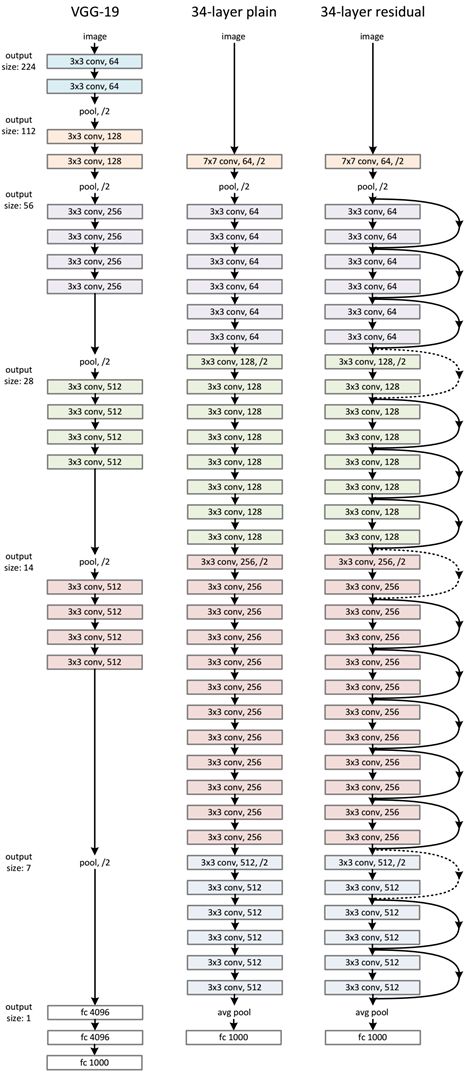
CNNs also often have pooling layers, which decrease the spatial size of features. By decreasing the size, pooling layers can decrease the computational power required to train and run the model. The two types of pooling are average pooling and max pooling. Average pooling outputs the average of the values in the pooling area, while max pooling outputs the biggest value in the pooling area.

After finishing all of the convolutions the data is fed into a normal fully connected neural network for the actual classification. The features must first be flattened into a one-dimensional array before being inputted into the fully connected layer. At this point, not much spatial information is lost, because the convolutional layers have already extracted high level features. At the end of the fully connected layers, the outputted data is put through a SoftMax function, giving the probability of different attributes.

**ResNet**

Residual neural networks (ResNets) is an improved CNN structure that ultilizes skip connections. The biggest advantage of ResNets are that they ease the problem of vanishing gradients. Without ResNets deeper CNNs would sometimes perform worse than shallower ones even without overfitting, but ResNets allow deeper nets to function and perform better.



The structure of ResNets are very similar to plain CNNs that were explained earlier, except at regular intervals some layers are skipped where their outputs are added to the outputs of previous layers.

This solves the issue of vanishing gradients. Usually, in nets with a lot of layers, the gradients farthest away from the loss function can become to small and vanish. This is because the gradient must be multiplied with all the gradients in the layers before it due to the chain rule. If the gradients before are close to 0, then after multiplying many gradients together, the value of the final gradient can become so small that it exceeds the floating point number precision limit and eventually becomes 0, making it impossible to optimize the weights in the layer as the gradient cannot be calculated. In ResNets, due to the use of skip connections, the gradients of the layer farthest away from the loss function can partially be calculated directly against the loss function, because the skip connections allow the first layer to affect the loss function directly through additions.

**How to train**

There are four steps for each batch of the training process: forward, calculating loss, backward and optimization.

Forward or forward propagation is the first step, where the training data in the batch is fed through the model and the output of the model is calculated.

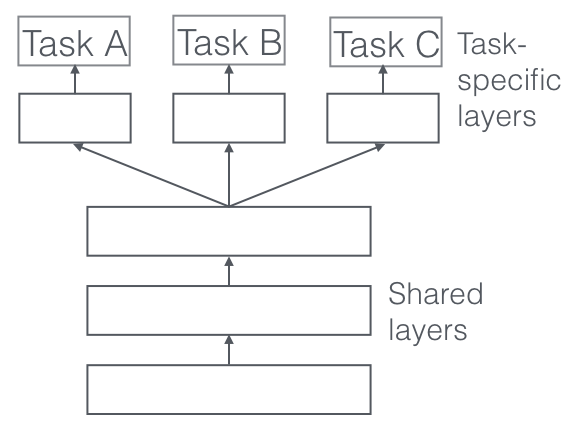
In the second step, calculating loss, the output from forward is compared to the expected output. The difference between the output of the model and the expected output is used to calculate the loss of the model.

In the third step, which is backward or backpropagation, the gradient of each parameter in the model against the loss function is calculated. As explained earlier, the chain rule is used to calculate the gradients of weights more than one layer away from the loss function, because the calculation for the output has functions inside functions.

Finally, the weights are adjusted by subtracting the learning rate multiplied by the gradient previously calculated.

The four steps are repeated for each batch in the training data, which is then run through the network more times until the network has finished training.

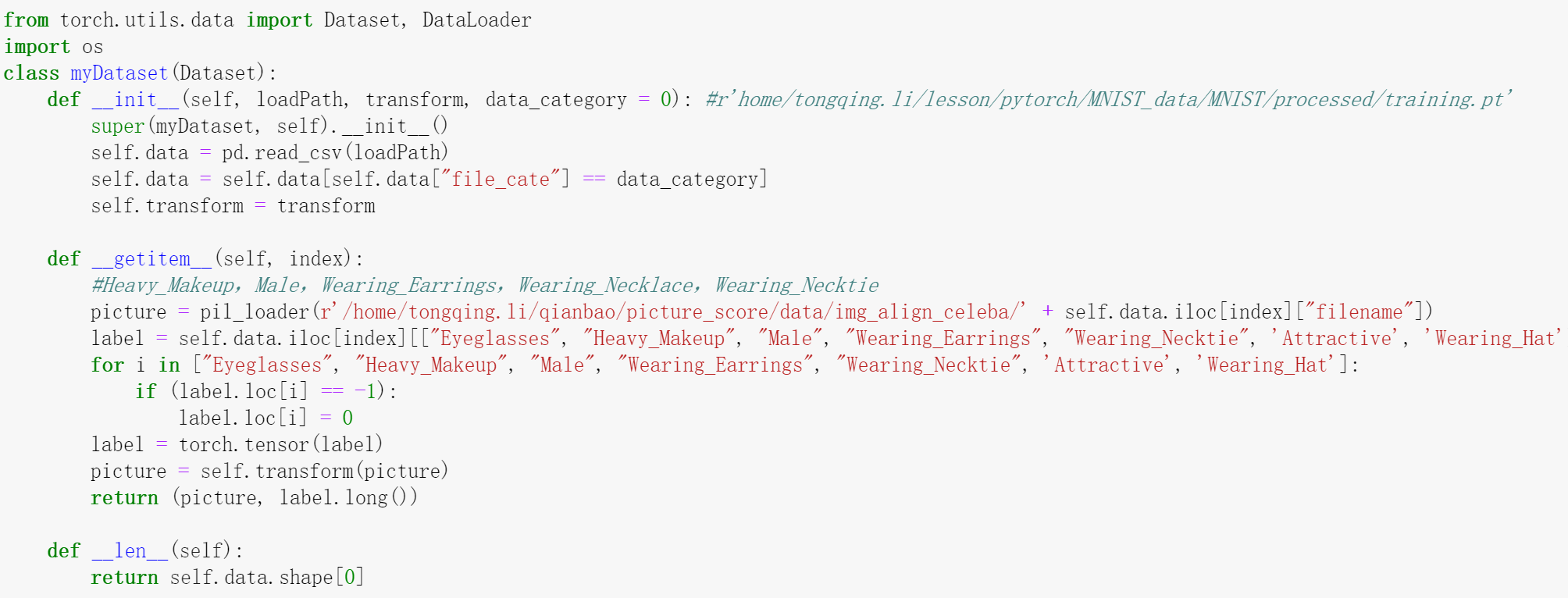
**Multi-task learning**



Multi-task learning is when a machine learning algorithm learns to do multiple tasks at the same time. In a usual neural net, all the outputs would be put into one SoftMax creating a list of probabilities of each feature. The feature with the highest probability would be the only one flagged as true. However, because this task requires that multiple features be extracted, we need multiple sets of outputs, each with its own SoftMax. In the case of this task, only two values would be put through each SoftMax: the probability that the feature is in the picture and the probability it is not. To do this, hard parameter sharing was used (shown to the right). In the final model, each feature was split after the convolutions and each feature had its own individual fully connected layer

**Python Implementation**

To create the model Pytorch was used. Pytorch is a deep learning python package made by Facebook. The main modules used are below.

The class myDataset is a custom dataset that inherits the map-style dataset class from Pytorch. The class is used to load data in and includes an image (the variable picture) and the labels attached to the image (the variable label).

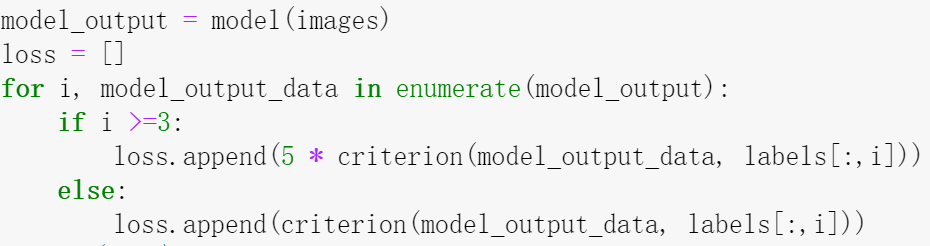


The function nn.Conv2d is used to make a convolution layer. In the example above, in\_planes is the number of input features, while out\_planes is the number of output features. The kernel\_size has been set to three which means that the convolution will be done through a kernel of size 3x3. As mentioned earlier, the stride is how much the kernel will move after completing a calculation. Padding is the extra area added around the input features to change the size of the output features. Finally the variable bias controls whether a bias will be used in the convolution (bias is the b in wx + b).



nn.MaxPool2d is the pytorch implementation for max pooling (explained earlier). In this example a kernel of size 3x3 is moved across the feature at two pixels at a time and takes the largest value in the kernel area of each stop and outputs it.





The example above shows how the loss is calculated for a multi-task neural net. In the example, the criterion used for calculating the loss of each individual task is nn.CrossEntropyLoss. For each task the loss is calculated using log\_ps, which is the output of the data and then it is appended to the loss list. Some of the losses have been multiplied by 5 for the model to train those as a priority since those tasks are more difficult to learn. The list of losses is later be summed up for backpropagation.



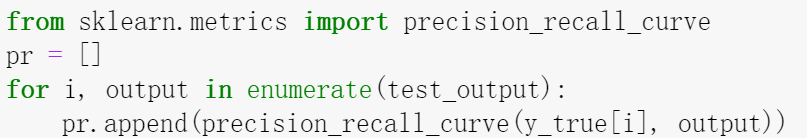
The function used for optimizing the model was torch.optim.SGD. SGD stands for stochastic gradient descent and the process is the same as explained in the final step of the how to train section.

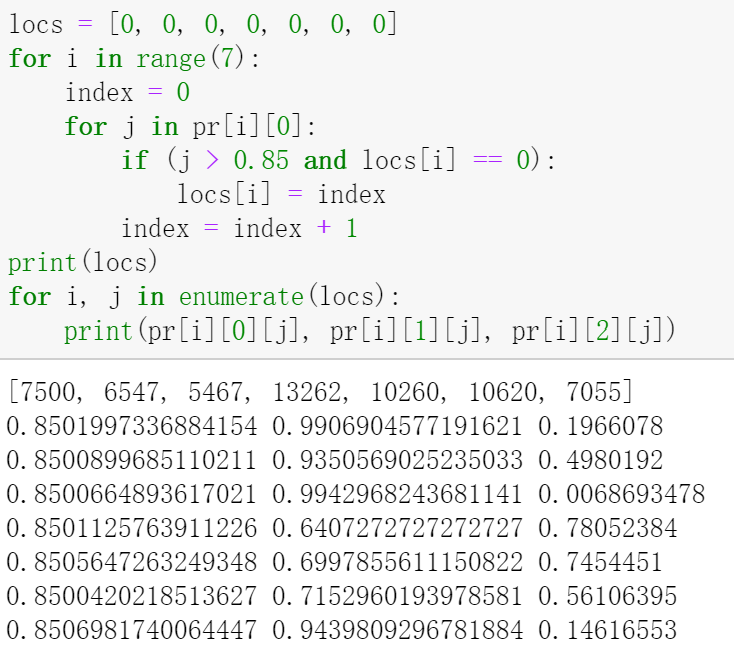


Optimizer.step() is then used when actually training the model to use SGD to update the weights.

After training the model, the precision and recall of the model must be calculated. Precision and recall are better measurements of the performance of the model than accuracy. This is because accuracy does not take into account how much of the data is positive. Because of this, if only 1% of the data was positive for a certain attribute and the neural net did not identify any of them correctly, the accuracy would still come out as 99%.

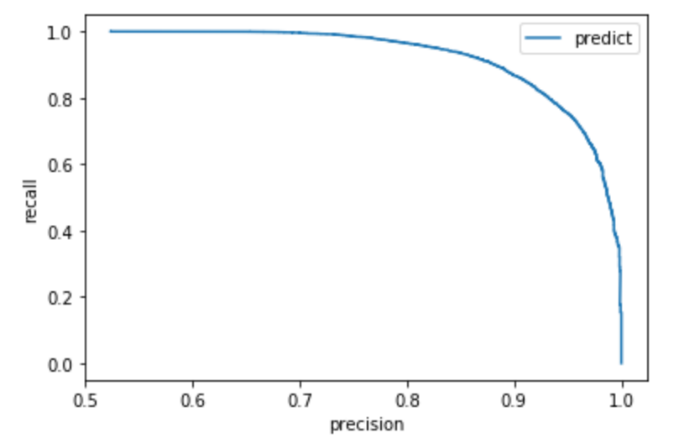
Precision is calculated using the formula tp/(tp+fp), where tp is true positives and fp is false positives. Precision measures if what the model evaluated to be positive is actually positive. Recall is calculated using the formula tp/(tp + fn). Recall measures the percentage of people with the selected attribute that the model can pick out.

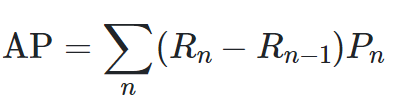




The precision and recall were calculated using sklearn. In the code above y\_true is a list containing the original labels and test\_output is what the model has outputted from the test data. After using precision\_recall\_curve() to calculate the precision and recall at different cutoff values for the probabilities, the data is added to the list pr[].

The code to the right finds the values for the recall and cutoff at a particular precision. In this example, the precision chosen is 85%. If the recall at this precision is good enough, the cutoff value can be used later to evaluate the actual users.

The data from the function precision\_recall\_curve can also be used to plot a graph which shows the precision against recall at all the different cutoffs. The larger the area under the graph, the better model is at identifying the attribute. The area can be calculated using the function average\_precision\_score. The area is calculated using the equation below where Pn and Rn are the precision and recalls at the nth cutoff.



**Experiment**

The dataset used to train the model was CelebA dataset which includes over 200,000 images with 40 attributes for each image. The data was split into three sections, training, validation and test. Training had 162,270 images while validation and test had around 20,000.

The precision recall curves were calculated for a Resnet50 model and a Resnet 18 model. The results are displayed to blow with one column showing recall when precision is 95% and the other showing recall when precision is 85%. The last column shows the area under the precision recall curves.

Resnet50

|  |  |  |  |
| --- | --- | --- | --- |
| Attribute | 95% precision | 85% precision | AP |
| Eyeglasses | 0.978 | 0.990 | 0.989 |
| Heavy Makeup | 0.753 | 0.935 | 0.961 |
| Necktie | 0.476 | 0.745 | 0.864 |

Resnet18

|  |  |  |  |
| --- | --- | --- | --- |
| Attribute | 95% precision | 85% precision | AP |
| Eyeglasses | 0.979 | 0.987 | 0.988 |
| Heavy Makeup | 0.785 | 0.938 | 0.967 |
| Necktie | 0.445 | 0.719 | 0.871 |

From the chart, it can be seen that the Resnet50 model is much better than Resnet18 at classifying neckties, while Resnet 18 is slightly better at classifying heavy makeup. The two models have similar performance when classifying eyeglasses. Because the it was more important for the model to be able to classify neckties, the Resnet50 model was used as it had better recall for neckties at 85% precision. The cutoff for eyeglasses was set to the cutoff for 95% precision while the other two were set for 85% because the model initially identified very few people (47 and 9 for necktie and heavy makeup respectively on user data).

The model was then used on a user dataset from X Financial, which includes around 114,000 people, each with a picture.

Using the adjusted cutoffs, users that the model flagged as having heavy makeup were 27.5% less likely to have overdue loans. Users that the model flagged as wearing a necktie were 17.3% less likely to have overdue loans, while users that were flagged as wearing glasses were 30.8% less likely to have overdue loans. This data is shown in the chart below. Flag = 0 is the percentage of people with overdue loans that the model has flagged with 0 and Flag = 1 is the percentage of people flagged with the attribute that have overdue loans.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Attribute | Flag = 0 | Flag = 1 | Ratio | % with attribute |
| Eyeglasses | 5.36% | 3.71% | 0.692 | 13.4% |
| Heavy Makeup | 5.16% | 3.75% | 0.725 | 1.7% |
| Necktie | 5.14% | 4.25% | 0.827 | 0.23% |

Despite adjusting the cutoffs for heavy makeup and neckties, the model still was not able to find many people with those attributes, as shown by the chart above in the % with attribute column. Because of this, only the attribute eyeglasses was added to the risk model using logistic regression. The logistic regression was trained using 20,178 users and tested using 57,203 users.

After eyeglasses was added to the model the number of people in the top 10% of risk levels which had overdue loans went from 19.42% to 19.72%. The rate for the top 20 % increased from 34.77 to 35.04. The K-S also increased from 21.67% to 22.24%, which means the model improved.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Metric | Original model | Modified model | Lift | Lift % |
| Top 10% | 19.42% | 19.72% | 0.3% | 1.54% |
| Top 20% | 34.77% | 35.04% | 0.27% | 0.77% |
| K-S | 21.67% | 22.24% | 0.57% | 2.63% |