Homework 1: The Very Basics

Reno Malanga, Christian Arndt
Behavioral Data Science

February 24, 2025

Problem 1

Mark all statements which are $\boxed{\text{FALSE}}$.

- [✓ NumPy arrays and TensorFlow tensors are fundamentally interchangeable and can always be used interchangeably in all TensorFlow operations without any explicit conversion.]
- [✓ Stochastic gradient descent (SGD) with a fixed learning rate always converges faster than Adam when training deep neural networks.]
- [✓ The Jacobian matrix of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ has shape (n, m).]
- [✓ Mini-batch gradient descent is practically never used in deep learning due to its high memory demands.]
- [✓ TensorFlow's automatic differentiation engine (autodiff) uses numerical differentiation to compute gradients with respect to variable nodes.]
- [✓ The second derivative of the loss function with respect to individual model parameters is always required for implementing gradient-based optimization methods.]
- The sigmoid function can be used both as an activation function in hidden layers of MLPs and as an output activation function for binary classification tasks.
- Setting `tf.random.set.seed(seed)` ensures deterministic behavior for TensorFlow's random functions.

Problem 2

PyTorch versus TensorFlow

Problem Setup

Read up the key differences between PyTorch and TensorFlow, explaining what they mean for users of the frameworks. Provide a snippet of how you would define the same fully connected network we used in class for ASD classification.

Problem Solution

Both PyTorch and TensorFlow have been used when developing deep learning networks, of course, there are some key differences between the two, each resulting in their own pros and cons. The first key difference is how the code is executed. Both libraries utilize computational graphs, however, the mechanism itself varies. TensorFlow's computational graph is defined as static whereas PyTorch is dynamic.

Since TensorFlow utilizes a static computational graph, it allows for more efficient training, but it does allow for dynamic graphs if the correct library is imported, which is a testament to its versatility. One key feature that TensorFlow has is data parallelism, where all of the code needs to be manually fine-tuned compared to PyTorch which works asynchronously from Python.

Another key difference between the two libraries is the type of work being implemented. PyTorch focuses more on research-based projects whereas TensorFlow looks more towards production.

TensorFlow vs PyTorch Deep Neural Networks

Below is an example of the neural network we trained in class implemented in both TensorFlow and PyTorch.

```
##################
### Tensorflow ###
##################

tf_nn_model = tf.keras.Sequential([
    tf.keras.Input(shape=(x_train.shape[1],)),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

# Specify the optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

# Compile the model
tf_nn_model.compile(optimizer=optimizer, loss="binary_crossentropy", \
            metrics=["accuracy"])

# Train the model
history = tf_nn_model.fit(x=x_train, y=y_train, batch_size=32, epochs=100)



##################
##### Torch ######
##################

# Model shape definition
class ASDClassifier(nn.Module):

    def __init__(self, input_dim: int):
        super().__init__()

        # Layers
        # Input and output layer
        self.input_layer = nn.Linear(input_dim, 64)
        self.output_layer = nn.Linear(64, 1)

        # Three hidden layers
        self.hidden1 = nn.Linear(64, 64)
        self.hidden2 = nn.Linear(64, 64)
        self.hidden3 = nn.Linear(64, 64)

        # Relevant activation functions
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Defining network structure
        x_1 = self.relu(self.input_layer(x))
        x_2 = self.relu(self.hidden1(x_1))
        x_3 = self.relu(self.hidden2(x_2))
        x_4 = self.relu(self.hidden2(x_3))
        x_5 = self.sigmoid(self.output_layer(x_4))
        return x_5

    # Instantiate the model
    torch_nn_model = ASDClassifier(x_train.shape[1])

    # Define the loss and optimizer
    loss_fn = nn.BCELoss()
    optimizer = optim.Adam(torch_nn_model.parameters(), lr=1e-3)

    # Define training parameters
    batch_size = 32
    epochs = 100

    # Training loop
    for epoch in range(epochs):

        # Shuffle the data
        x_shuffled, y_shuffled = sklearn.shuffle(x_train, y_train)

        # Convert dataframe to tensors
        x_train_as_tensor = torch.tensor(x_shuffled.values)
        y_train_as_tensor = torch.tensor(y_shuffled)

        for batch in range(0, x_train.shape[0], batch_size):

            # Grab batch from data
            xbatch = x_train_as_tensor[batch:batch+batch_size]
            ybatch_true = y_train_as_tensor[batch:batch+batch_size].unsqueeze(1)

            # Get model predictions
            ybatch_pred = torch_nn_model(xbatch)

            # Compute loss
            loss = loss_fn(ybatch_pred, ybatch_true)

            # Zero gradient
            optimizer.zero_grad()

            # Do backpropagation
            loss.backward()

            # Update weights
            optimizer.step()
        # Report each epoch
        print(f"Finished training epoch {epoch + 1}; loss {loss}")
```

Problem 3

Gradient Descent

Problem Setup

1. What is the gradient of a multi-variable function and what information does it provide? What is the goal of *gradient descent*? For the function
$$C(\theta_1, \theta_2) = 2\theta_1^2 + 3\theta_1\theta_2 \tag{1}$$
perform the three manual steps of gradient descent for a step size α of your choosing. Initialize the vector $\theta = (\theta_1, \theta_2)$ with the first and second digit of your birthday, for instance, January 1st: $\theta^{(0)} = (0, 1)$, May 16: $\theta^{(0)} = (1, 6)$.

2. For the multivariate function $u(x, y) = x^2 - y^2$ and $w(x, y) = 2xy + e^x$, find the Jacobian matrix **J** and evaluate it at the points (1, 1) and (0, 0).

Problem Solution

1. The gradient of a function is a vector which points in the direction of the function's maximum slope at a given point. If a function f has an n-dimensional input, the gradient is a vector in n-dimensional space, where each vector component is the partial derivative of the function over that input variable. Using the information the gradient provides, we can use the gradient to 'walk along' the surface of a function to find maxima (with gradient ascent) and minima (with gradient descent). We usually use gradient descent to minimize a loss function over model parameters.

   Let's consider the function
$$C(\theta_1, \theta_2) = 2\theta_1^2 + 3\theta_1\theta_2 \tag{2}$$
   First, let's find the gradient. To do this, we need to calculate the partial derivative of C with respect to both input variables. These partial derivatives will give us the components of the gradient vector.

$$\frac{\partial C}{\partial \theta_1} = 4\theta_1 + 3\theta_2$$
$$\frac{\partial C}{\partial \theta_2} = 3\theta_1$$

This gives us a gradient of:

$$\nabla C(\theta_1, \theta_2) = \begin{bmatrix} 4\theta_1 + 3\theta_2 \\ 3\theta_1 \end{bmatrix}$$

Now we have a formula for the gradient; what remains is to perform three steps of the gradient descent algorithm. Let $\alpha = 1$.

(a) First, we randomly initialize $\theta_0 = (1, 9)$

(b) Next, we perform gradient descent with three steps:

i. First, calculate the gradient at $(1, 9)$:

$$\nabla C(1, 9) = \begin{bmatrix} 31 \\ 3 \end{bmatrix}$$

Next, subtract the gradient times α from θ_0 to get θ_1. We're using $\alpha = 1$, so we just subtract the gradient:

$$\begin{aligned} \theta_1 &= \begin{bmatrix} 1 \\ 9 \end{bmatrix} - \begin{bmatrix} 31 \\ 3 \end{bmatrix} \\ &= \begin{bmatrix} -30 \\ 6 \end{bmatrix} \end{aligned}$$

ii. Repeat the same steps, but with θ_1:

$$\begin{aligned} \nabla C(-30, 6) &= \begin{bmatrix} -102 \\ -90 \end{bmatrix} \\ \theta_2 &= \begin{bmatrix} -30 \\ 6 \end{bmatrix} - \begin{bmatrix} -102 \\ -90 \end{bmatrix} \\ &= \begin{bmatrix} 72 \\ 96 \end{bmatrix} \end{aligned}$$

iii. Finally, do the same thing again with θ_2:

$$\begin{aligned} \nabla C(72, 96) &= \begin{bmatrix} 576 \\ 216 \end{bmatrix} \\ \theta_3 &= \begin{bmatrix} 72 \\ 96 \end{bmatrix} - \begin{bmatrix} 576 \\ 216 \end{bmatrix} \\ &= \begin{bmatrix} -504 \\ -120 \end{bmatrix} \end{aligned}$$

iv. This gives us our output:

$$\theta_3 = \begin{bmatrix} -504 \\ -120 \end{bmatrix}$$

2. The Jacobian matrix of the described function can be defined as follows:

$$\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} \end{bmatrix}$$

Let us plug in the evaluated partials:

$$\mathbf{J} = \begin{bmatrix} 2x & 2y \\ 2y + e^x & 2x \end{bmatrix}$$

Finally, let us evaluate the above Jacobian at the points $(1,1)$ and $(0,0)$ by plugging those points into the partials and then finding the determinant of the resultant distribution, remembering that the determinant of a matrix A is defined as follows:

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc$$

(a) First, for the point $(1,1)$:

$$\mathbf{J}_{1,1} = \begin{bmatrix} 2 & 2 \\ 2+e & 2 \end{bmatrix}$$

To evaluate further, we find the determinant:

$$\begin{aligned} \det(\mathbf{J}_{1,1}) &= 2 * 2 - 2 * (2 - e) \\ &= 4 - 4 + 2e \\ &= 2e \end{aligned}$$

(b) Next, for the point $(0,0)$:

$$\mathbf{J}_{0,0} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

To evaluate further, we find the determinant:

$$\begin{aligned} \det(\mathbf{J}_{0,0}) &= 0 * 0 - 0 * (0 - e) \\ &= 0 \end{aligned}$$

Problem 4

Maximum Likelihood Estimation Continued

Problem Setup

In the last homework, you derived the maximum likelihood estimate of the mean under a Gaussian data model. In this exercise, you will explore a deeper connection between maximum likelihood estimation (MLE) and information theory, ultimately leading to an understanding of MLE as distribution matching.

Let's restate our goal first: we want to minimize the difference between our data model $p(x \mid \theta)$ and the assumed data-generating distribution $p^*(x)$. But how do we express the notion of difference or *distance* between distributions? Fortunately, information theory has the answer, which comes in the form of the Kullback-Leibler (KL) divergence, defined for continuous densities p and q as:

$$\mathrm{KL}(p \| q) = \int p(x) \log \left(\frac{p(x)}{q(x)} \right) dx \tag{3}$$

The KL divergence has a very neat property: it evaluates to 0 if and only if the two densities are the same, that is, $p = q$ (you can try to show this as a bonus exercise). To make the connection between MLE and the KL divergence, your first task is to show that minimizing the KL divergence leads directly to the familiar and general MLE formulation:

$$\theta_{\text{MLE}} = \underset{\theta}{\text{argmin}}\ \mathrm{KL}(p^*(x) \| p(x \mid \theta)) \tag{4}$$

Did you notice something about the order of arguments in the KL? Your second task is to show that the KL is **non-symmetric**, that is, swapping the order of arguments in Eq. 2 results in a different value whenever $p \neq q$. In other words, you will show that, strictly speaking, the KL is not a proper distance, hence the wording *divergence*.

Problem Solution

KL Divergence and MLE

Let us first state the definition of MLE given data $\mathbf{X} = \{x_1, x_2, \dots, x_N\}$:

$$\theta_{\text{MLE}} = \underset{\theta}{\text{argmax}} \prod_{i=1}^{N} p(x_i \mid \theta)$$

In plain terms, θ_{MLE} is the set of parameters which maximizes the likelihood across all data points.

KL divergence is a measure of how far apart two distributions are. $\mathrm{KL}(p^*(x) \| p(x \mid \theta))$, then, is how far the conditional distribution $p(x \mid \theta)$ is from the true distribution, $p^*(x)$. What we're trying to show is that the θ which minimizes the divergence of $p(x \mid \theta)$ from $p^*(x)$ is the same as the θ which maximizes the likelihood of $p(x \mid \theta)$. This makes intuitive sense; the closer our parameterized distribution is to reality, the higher the probability should be over the data.

With the intuition established, let's move on to the proof.

First, let's consider the formula for the expectation of a continuous random variable:

$$\mathbb{E}_{p(x)}[x] = \int p(x) * x * dx$$

From this, we can reframe the KL divergence as the expectation of a random variable y where $y = \log(p(x)/q(x))$. In other words:

$$\mathrm{KL}(p^*(x) \| p(x \mid \theta)) = \mathbb{E}_{p^*(x)} \left[\log \left(\frac{p^*(x)}{p(x \mid \theta)} \right) \right]$$

Let us re-write the argmin formula this way and then rearrange and, because it's an $\underset{\theta}{\operatorname{argmin}}$ over θ, drop terms unrelated to θ:

$$\begin{aligned}
\mathrm{KL}(p^*(x) \| p(x \mid \theta)) &= \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{p^*(x)} \left[\log \left(\frac{p^*(x)}{p(x \mid \theta)} \right) \right] \\
&= \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{p^*(x)} \left[\log(p^*(x)) - \log(p(x \mid \theta)) \right] \\
&= \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{p^*(x)} \left[-\log(p(x \mid \theta)) \right]
\end{aligned}$$

Our goal now is to transform the above expression into the expression for θ_{MLE}.

Expectation is linear, so we can move the negative sign outside. Then we can flip the sign and turn the $\underset{\theta}{\operatorname{argmin}}$ into an $\underset{\theta}{\operatorname{argmax}}$:

$$\begin{aligned}
&= \underset{\theta}{\operatorname{argmin}} - \mathbb{E}_{p^*(x)} \left[\log(p(x \mid \theta)) \right] \\
&= \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{p^*(x)} \left[\log(p(x \mid \theta)) \right]
\end{aligned}$$

Now we need to get rid of the expectation. If we were to plug the above into the formula for expectation, though, we'd end up with p^* inside the integral; that isn't good, because p^* isn't found in the MLE definition. We can get around this using the Monte Carlo approximation of an expectation, which states that

$$\mathbb{E}_{p^*(x)}[f(x)] \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

Note that the approximation can get arbitrarily close to the true expectation as n increases. Plugging $\log(p(x \mid \theta))$ in for $f(x)$ in the above, we get:

$$\begin{aligned}
&= \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{p^*(x)} \left[\log(p(x \mid \theta)) \right] \\
&\approx \underset{\theta}{\operatorname{argmax}} \frac{1}{n} \sum_{i=1}^{n} \log(p(x_i \mid \theta))
\end{aligned}$$

The leading factor of $1/n$ is a regularization factor; this won't affect the $\underset{\theta}{\operatorname{argmax}}$, so we can simply drop it.

$$\approx \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} \log(p(x_i \mid \theta))$$

We're now maximizing the log-likelihood, which is equivalent to maximizing the likelihood, so the proof is over. Still, if we want to get it into the MLE-form we defined above, we can move the log outside the sum (to turn it into a product) and then take advantage of the fact that $\underset{\theta}{\operatorname{argmax}} f(x) = \underset{\theta}{\operatorname{argmax}} \log(f(x))$ to drop the log:

$$\begin{aligned}
&\approx \underset{\theta}{\operatorname{argmax}} \log\left(\prod_{i=1}^{n} p(x_i \mid \theta)\right) \\
&\approx \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{n} p(x_i \mid \theta) \\
&\approx \theta_{\text{MLE}}
\end{aligned}$$

As we noted before, the approximation becomes arbitrarily close to the truth as the size of our sample increases.

KL Divergence and Symmetry (or lack thereof)

To prove that KL divergence is non-symmetric, we'll define a counterexample. Consider the two PMFs p and q:

$$p(x) = \begin{cases} .5 & \text{if} \quad x = 1 \\ .5 & \text{if} \quad x = 2 \end{cases}$$

$$p(x) = \begin{cases} .25 & \text{if} \quad x = 1 \\ .75 & \text{if} \quad x = 2 \end{cases}$$

Let us now consider $\text{KL}(p \| q)$ and $\text{KL}(q \| p)$. Note that these are PMFs, not PDFs, so we'll be using a sum instead of an integral.

$$\begin{aligned}
\text{KL}(p \| q) &= p(x=1) \log\left(\frac{p(x=1)}{q(x=1)}\right) + p(x=2) \log\left(\frac{p(x=2)}{q(x=2)}\right) \\
&= .5 * \log\left(\frac{.5}{.25}\right) + .5 * \log\left(\frac{.5}{.75}\right) \\
&= .5 * \log(2) + .5 * \log(2/3) \\
&\approx 0.062 \\
\text{KL}(q \| p) &= q(x=1) \log\left(\frac{q(x=1)}{p(x=1)}\right) + q(x=2) \log\left(\frac{q(x=2)}{p(x=2)}\right) \\
&= .25 * \log(.5) + .75 * \log(3/2) \\
&\approx 0.057
\end{aligned}$$

This example shows that there exist distributions p and q such that $\text{KL}(p \| q) \neq \text{KL}(q \| p)$.

Bonus exercise

We introduce an additional constraint, that both p and q are smooth functions, for two reasons:

- We don't know how to think about integration for arbitrary non-smooth functions
- We need to make the assumption that, given $x = n : p(x) \neq 0$ or $q(x) \neq 0$, there is some ϵ for which all $x \in [n - \epsilon, n + \epsilon], p(x) \neq 0$.

We also ignore any potential messiness where $q = 0$. We believe this is justified because bonus exercises require less rigor than main questions.

First, let us prove that the KL divergence is 0 if $p = q$.

If $p = q$, which is to say, if $p(x) = q(x)$ for all x, then $p(x)/q(x) = 1$ for all x. We know that $\log(1) = 0$. The function inside the integral has $\log(p(x)/q(x))$ as a factor. Putting these three facts together, we can see that if $p = q$, the function inside the integral will evaluate to 0 everywhere, meaning there is no area under the curve and the integral (and Kullback-Leibler convergence) will also evaluate to 0.

Now, let us prove the the KL divergence is non-zero if $p \neq q$.

Let us first state that, if $p \neq q$, $\exists a, \epsilon : \forall x \in [a - \epsilon, a + \epsilon] : p(x) \neq 0, p(x) \neq q(x)$.

The first half of the above, that there is some range of inputs $[a - \epsilon, a + \epsilon]$ for all of which $p(x) \neq 0$, is trivial. p is a PDF, meaning it must integrate to 1; we also know that p is smooth based on our assumption. This forces an interval over the domain for which the output $p(x)$ is non-zero.

Now we need to add in the second half; that is, we need to show that on top of $p(x) \neq 0$ for all $x \in [a - \epsilon, a + \epsilon]$, it is also true that $p(x) \neq q(x)$. We know that p and q are both PDFs, meaning they both non-negative and integrate to 1. We also know that $p \neq q$, meaning $\exists x(p(x) \neq q(x))$.

Assume that $\forall x, p(x) \neq 0 : p(x) = q(x)$. Given $p \neq q$, we know that $\exists x : p(x) \neq q(x)$. Putting these two together, we can see that $\exists d : p(d) = 0, q(d) \neq 0$. We know from our assumption of smoothness that this means there is some range around d, $[d - \epsilon, d + \epsilon]$ for all of which $q \neq 0$. Let the non-zero integral $\int q(x)dx$ over this range be i. Because we've assumed $\forall x, p(x) \neq 0 : p(x) = q(x)$, this means that $\int q(x)dx \geq i + \int p(x)dx$. Given that both p and q are PDFs, such an inequality is impossible.

Therefore it must be true that, if $p \neq q$, $\exists a, \epsilon : \forall x \in [a - \epsilon, a + \epsilon] : p(x) \neq 0, p(x) \neq q(x)$.

Given $x : p(x) \neq 0, p(x) \neq q(x)$, and ignoring the divide-by-zero case where $q(x) = 0$, we can see that the function under the integral will evaluate to a non-zero number. Our assumption of smoothness means this non-zero evaluation will hold true for some range around the given x, meaning the entire KL divergence will evaluate to something other than 0.

Having proved the implication in both directions, we can see that $\text{KL}(p\|q) = 0 \iff p = q$.

Problem 5

Simple Mixture-of-Experts (MoE) Approach

Problem Setup

In previous classes, we fitted three distinct models for ASD classification in our envisaged "app":

1. A logistic regression model as a baseline.
2. A rule-based decision model using the clinical score.
3. A fully connected neural network classifier.

In this exercise, you will explore an alternative approach that is often a strong baseline for classification tasks on tabular data.

Step 1: Background Reading

First, familiarize yourself with **Decision Trees** and the **Random Forest Algorithm** using the following non-limiting resources:

- **YouTube**: Decision Trees and Random Forests (Video)
- **Kaggle**: Random Forest Classifier Tutorial

Step 2: Constructing Meta-Features

Next create new feature sets for the training and test data. These feature sets should consist of the predictions from the three models above. This will result in a new dataset with three columns, each representing the predictions of one of the models.

Step 3: Training the Random Forest Classifier

Use the `sklearn` implementation of the Random Forest classifier to fit a model using the predictions from the three models as input features. Evaluate its accuracy on the test set. This effectively learns a "weighting" of the three individual classifiers.

Step 4: Comparing Performance

Finally, compare the performance of this mixture-of-experts approach to a straightforward application of a Random Forest classifier trained directly on the original dataset. Summarize your findings, highlighting any improvements or differences in classification accuracy.

Problem Solution

Random Forest Classifier Build Notes

We constructed the meta-features by using each of the three models (the linear regression, clinical test, and neural network) on both the test and train data and concatenating the resulting columns into a single data frame. We then exported this dataframe, loaded it, and used `sk.model_selection.train_test_spl` to create our training and test sets.

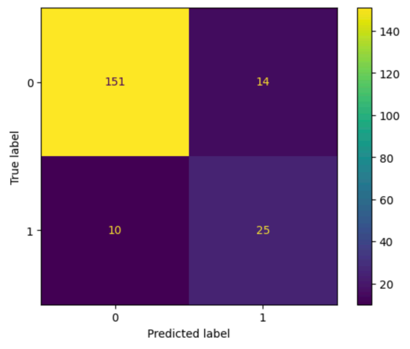
One result of this procedure is that the train and test sets we used for the MOE random forest classifier won't contain the same data points as those in the train and test sets as defined by the data files from LMS.

Model Performance

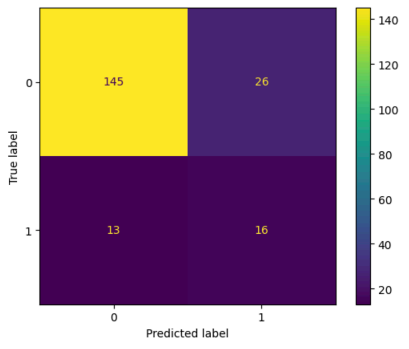
On every metric, the MOE random forest classifier performed better than the random forest classifier trained on the original data. The MOE model's accuracy is .88, while the standard model's accuracy is only .805. The base rate across the entire sample (including both test and train data) is 0.81, which is slightly higher than the standard model's accuracy.

Below are the confusion matrices for each model:

MOE Model



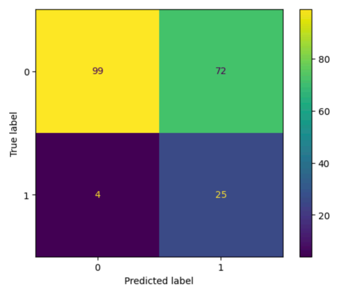
Standard Model



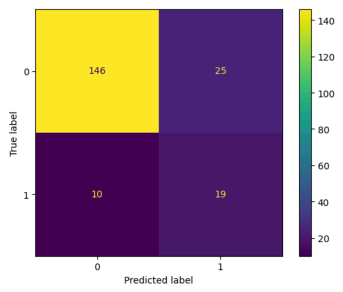
As we can see, the MOE model performs better than the standard model in every single quadrant; it has more true positives and negatives, and fewer false positives and negatives.

Below are the confusion matrices for each of the constituent ’expert’ models we used in creating the MOE classifier:

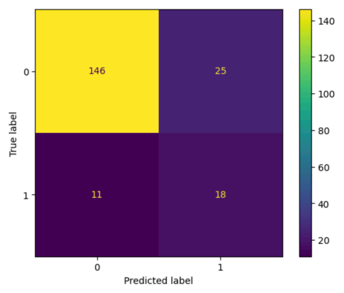
Clinical Model



Linear Regression



Neural Network



As we can see, it also outperforms each of these models. In fact, across all of these models, the only quadrant which is not strictly outperformed by the MOE classifier is the lower-right quadrant of the clinical model; here, the MOE classifier and the clinical test tie. But a quick look at the rest of the clinical test confusion matrix makes it immediately obvious that our MOE model is significantly better overall.