IDA-ML I: Project Presentation

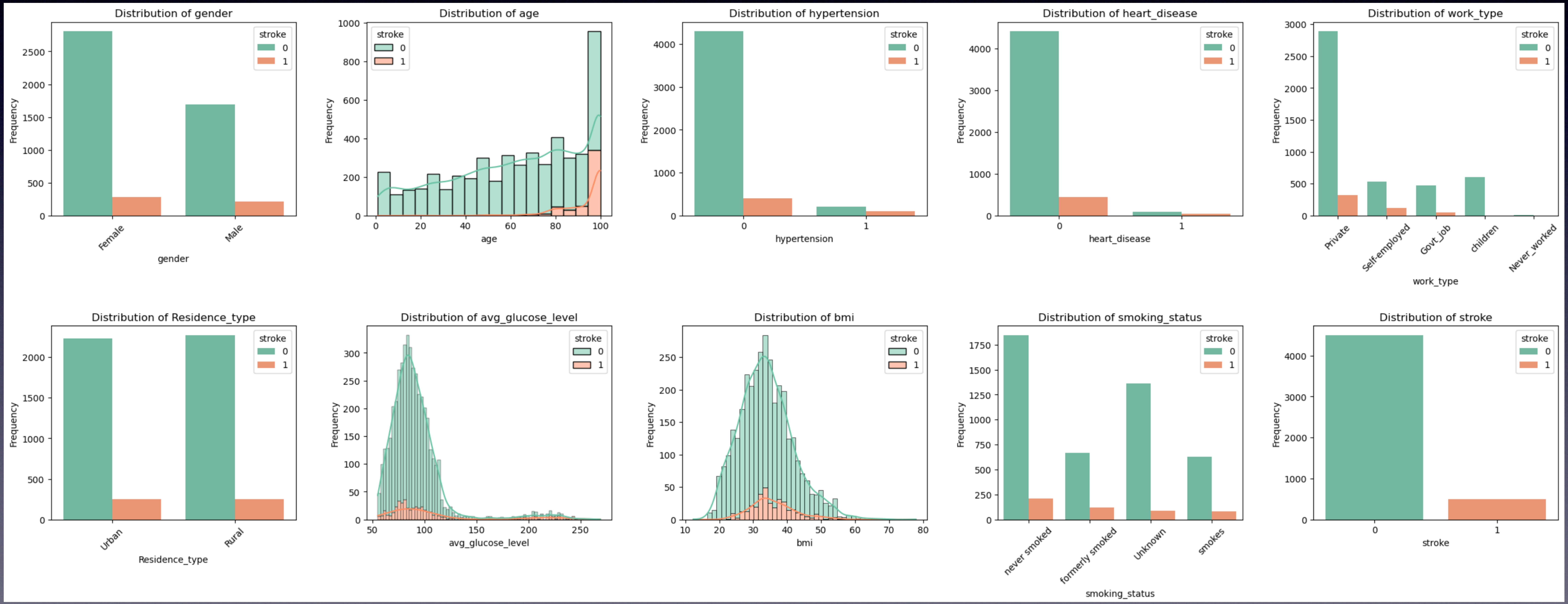
Project: Binary Stroke Classification
Examinee: Paul Utsch
Date: August 7, 2024

Table of Contents

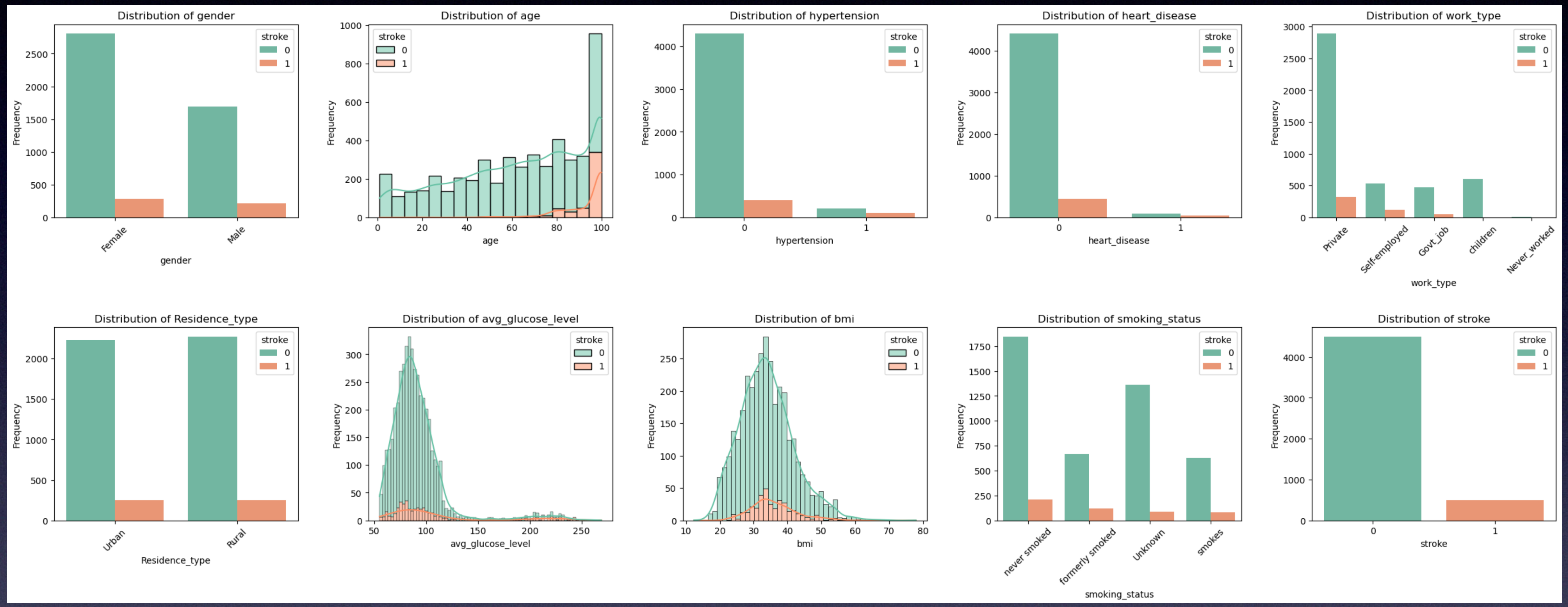
1. Problem Statement

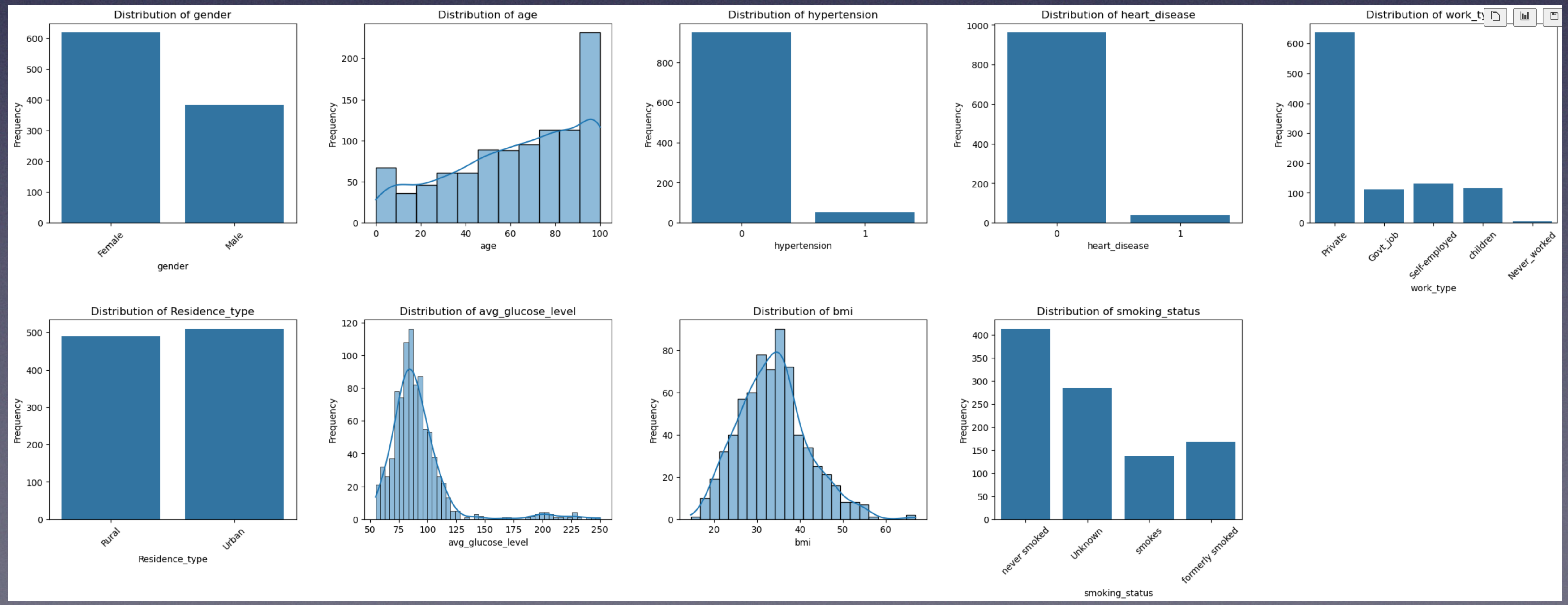
- **Supervised Binary Classification Problem**
- **Predict stroke patients** based on medical and personal data
- **Data Set**:
  - **5000 samples** in train set, **1000 samples** in test set
  - **fictional data**
  - **source**: https://www.kaggle.com/competitions/ida-ml-1-challenge-summer23/overview

2. Data Analysis & Preprocessing

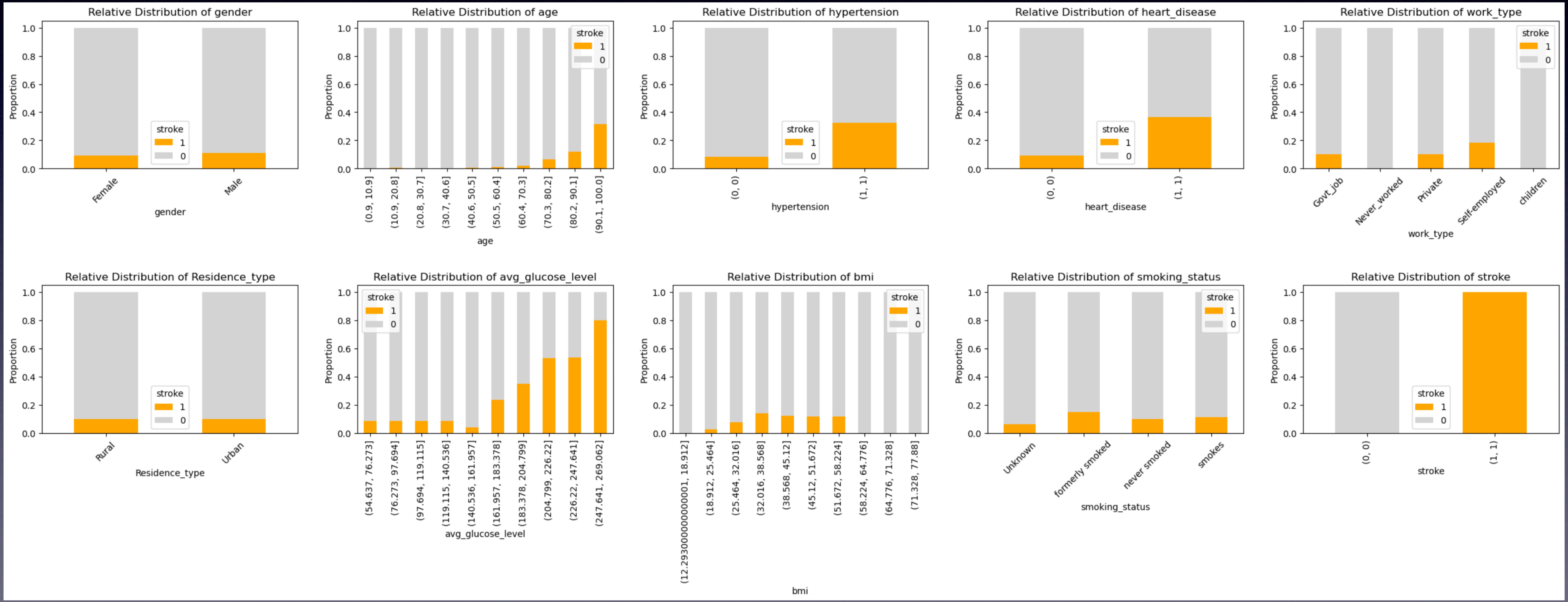


2. Data Analysis & Preprocessing





2. Data Analysis & Preprocessing



2. Data Analysis & Preprocessing

```
2024-08-04 16:14:33.772 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column gender: 0 / 5000
2024-08-04 16:14:33.772 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column age: 0 / 5000
2024-08-04 16:14:33.773 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column hypertension: 0 / 5000
2024-08-04 16:14:33.774 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column heart_disease: 0 / 5000
2024-08-04 16:14:33.775 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column work_type: 0 / 5000
2024-08-04 16:14:33.775 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column Residence_type: 0 / 5000
2024-08-04 16:14:33.776 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column avg_glucose_level: 393 / 5000
2024-08-04 16:14:33.776 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column bmi: 1552 / 5000
2024-08-04 16:14:33.777 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column smoking_status: 1451 / 5000
2024-08-04 16:14:33.778 | INFO     | src.data_understanding.data_exploration:print_na:133 - Number of NaN values in column stroke: 0 / 5000
2024-08-04 16:14:33.779 | INFO     | src.data_understanding.data_exploration:print_na:138 - Number of data points with NaN values: 2756 / 5000
```

2. Data Analysis & Preprocessing

- **Handling imbalanced data**:
  - Use **weighted cross-entropy loss** as loss function
- **Data preprocessing**:
  - Missing values: **Replace missing values with median**
  - Numerical representation: **Transform discrete features** into binary / one-hot-encoded features
  - **min-max-normalize continuous features**, range [0, 1]

3. Machine Learning Methods

- **Naive Implementation** as baseline model
  - random predictions based on target distribution in training set (should result in AUC of ~0.5)
- **Logistic Regression** as linear model
  - train on polynomial features (d=2) and principal components, effectively enabling some degree of non-linear separation with regards to the original data
- **Neural Network** as non-linear model
  - allow model to derive relevant non-linear relations on its own

⇒ non-linear separation: polynomials / PCs vs. Neural Network architecture

```python
class NaiveBaseline:
    def __init__(self):
        super().__init__()
        self.model = None
        self.p_true = 0.0

    def fit(
        self,
        X_train: npt.ArrayLike,
        y_train: npt.ArrayLike,
        X_val: npt.ArrayLike,
        y_val: npt.ArrayLike,
    ):
        self.p_true = y_train.mean()

        val_preds = self.predict(X_val)
        val_accuracy = (val_preds == y_val).mean()
        print(f"Validation accuracy: {val_accuracy}")

    def predict(self, X: ArrayLike) -> ndarray:
        """returns a random prediction based on the base probability of the true class, without any training"""
        return np.random.choice([0, 1], size=len(X), p=[1 - self.p_true, self.p_true])

    def predict_proba(self, X: ArrayLike) -> ndarray:
        return np.full(len(X), self.p_true)
```

```python
class NaiveBaseline:
    def __init__(self):
        super().__init__()
        self.model = None
        self.p_true = 0.0

    def fit(
        self,
        X_train: npt.ArrayLike,
        y_train: npt.ArrayLike,
        X_val: npt.ArrayLike,
        y_val: npt.ArrayLike,
    ):
        self.p_true = y_train.mean()

        val_preds = self.predict(X_val)
        val_accuracy = (val_preds == y_val).mean()
        print(f"Validation accuracy: {val_accuracy}")

    def predict(self, X: ArrayLike) -> ndarray:
        """returns a random prediction based on the base probability of the true class, without any training"""
        return np.random.choice([0, 1], size=len(X), p=[1 - self.p_true, self.p_true])

    def predict_proba(self, X: ArrayLike) -> ndarray:
        return np.full(len(X), self.p_true)
```

```python
class BinaryLogisticRegression(object):
    name = "Binary Logistic Regression"

    def __init__(
        self,
        n_features: int,
        epochs: int = 20,
        learning_rate: float = 0.1,
        batch_size: int = 8,
        lambda_reg: float = 0.1,
        regularization: str = "L2",
    ):
        """
        X: (n_samples, n_features)
        Y: (n_samples,)

        self.W: (n_features, 1)
        self.B: (1,)
        """
        self.n_features = n_features
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.batch_size = batch_size        You, last week • feat: own mode
        self.lambda_reg = lambda_reg
        self.regularization = regularization

        self.class_weights = np.zeros(2)

        # initialize weights and bias to zeros
        self.W = np.zeros(n_features)
        self.B = 0.0

    def forward(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Compute forward pass of the logistic regression model.

        Y_hat: (n_samples,)
        """
        z = np.dot(X, self.W) + self.B
        Y_hat = sigmoid(z)
        return Y_hat

    def predict(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Create a prediction matrix with `self.forward()`

        pred: (n_samples,)
        """
        y_hat = self.forward(X)
        pred = np.where(y_hat >= 0.5, 1, 0)

        return pred

    def predict_proba(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Predict probabilities for the input data.

        Y_hat: (n_samples,)
        """
        Y_hat = self.forward(X)
        return Y_hat
```

class BinaryLogisticRegression(object):

```python

    def _backward(
        self,
        X: npt.ArrayLike,
        Y: npt.ArrayLike,
        Y_hat: npt.ArrayLike,
    ) -> Tuple[npt.ArrayLike, npt.ArrayLike]:
        """
        Compute back propagation for logistic regression.
        """
        batch_size = X.shape[0]

        # derivative of the loss with respect to the pre-activation of the output layer
        # equivalent to d_L_d_z = Y_hat - Y in non-weighted binary cross-entropy
        d_L_d_z = delta_weighted_bce(Y_hat, Y, self.class_weights)

        d_L_d_W = np.dot(X.T, d_L_d_z) / batch_size
        d_L_d_B = np.sum(d_L_d_z, axis=0) / batch_size

        if self.regularization == "L1":
            # add L1 regularization derivative
            d_L_d_W += self.lambda_reg * np.sign(self.W)
        else:
            # add L2 regularization derivative
            d_L_d_W += self.lambda_reg * self.W

        return d_L_d_W, d_L_d_B
```

class BinaryLogisticRegression(object):

```python
    def fit(
        self,
        X: npt.ArrayLike,
        Y: npt.ArrayLike,
        X_val: npt.ArrayLike,
        Y_val: npt.ArrayLike,
        plot: bool = False,
    ):
        """
        Fit the logistic regression model to the training data.
        """
        n = X.shape[0]

        self.class_weights = compute_class_weight(
            class_weight="balanced", classes=np.array([0, 1]), y=Y
        )

        losses_train = []
        losses_val = []

        for epoch in range(self.epochs):
            epoch_loss_train = 0.0
            for i in range(0, n, self.batch_size):
                X_batch = X[i : i + self.batch_size]
                Y_batch = Y[i : i + self.batch_size]

                Y_hat = self.forward(X_batch)
                d_L_d_W, d_L_d_B = self._backward(X_batch, Y_batch, Y_hat)

                self.W -= self.learning_rate * d_L_d_W
                self.B -= self.learning_rate * d_L_d_B

                batch_loss = weighted_binary_cross_entropy_loss(
                    Y_hat, Y_batch, self.class_weights
                )
                epoch_loss_train += batch_loss * len(Y_batch)

            loss_train = epoch_loss_train / n

            Y_hat_val = self.forward(X_val)
            loss_val = weighted_binary_cross_entropy_loss(
                Y_hat_val, Y_val, self.class_weights
            )

            losses_train.append(loss_train)
            losses_val.append(loss_val)

            if plot:
                print(f"Epoch {epoch}: Train Loss: {loss_train}, Val Loss: {loss_val}")

        if plot:
            plt.plot(losses_train, label="Training Loss")
            plt.plot(losses_val, label="Validation Loss")
            plt.xlabel("Number of epochs")
            plt.ylabel("Loss")
            plt.title("Training and Validation Loss")
            plt.legend()
            plt.show()
            print("Training complete.")
```

```python
class BinaryLogisticRegression(object):
    name = "Binary Logistic Regression"

    def __init__(
        self,
        n_features: int,
        epochs: int = 20,
        learning_rate: float = 0.1,
        batch_size: int = 8,
        lambda_reg: float = 0.1,
        regularization: str = "L2",
    ):
        """
        X: (n_samples, n_features)
        Y: (n_samples,)

        self.W: (n_features, 1)
        self.B: (1,)
        """
        self.n_features = n_features
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.batch_size = batch_size        You, last week • feat: own mode
        self.lambda_reg = lambda_reg
        self.regularization = regularization

        self.class_weights = np.zeros(2)

        # initialize weights and bias to zeros
        self.W = np.zeros(n_features)
        self.B = 0.0

    def forward(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Compute forward pass of the logistic regression model.

        Y_hat: (n_samples,)
        """
        z = np.dot(X, self.W) + self.B
        Y_hat = sigmoid(z)
        return Y_hat

    def predict(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Create a prediction matrix with `self.forward()`

        pred: (n_samples,)
        """
        y_hat = self.forward(X)
        pred = np.where(y_hat >= 0.5, 1, 0)

        return pred

    def predict_proba(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Predict probabilities for the input data.

        Y_hat: (n_samples,)
        """
        Y_hat = self.forward(X)
        return Y_hat
```

```python
class BinaryLogisticRegression(object):

    def _backward(
        self,
        X: npt.ArrayLike,
        Y: npt.ArrayLike,
        Y_hat: npt.ArrayLike,
    ) -> Tuple[npt.ArrayLike, npt.ArrayLike]:
        """
        Compute back propagation for logistic regression.
        """
        batch_size = X.shape[0]

        # derivative of the loss with respect to the pre-activation of the output layer
        # equivalent to d_L_d_z = Y_hat - Y in non-weighted binary cross-entropy
        d_L_d_z = delta_weighted_bce(Y_hat, Y, self.class_weights)

        d_L_d_W = np.dot(X.T, d_L_d_z) / batch_size
        d_L_d_B = np.sum(d_L_d_z, axis=0) / batch_size

        if self.regularization == "L1":
            # add L1 regularization derivative
            d_L_d_W += self.lambda_reg * np.sign(self.W)
        else:
            # add L2 regularization derivative
            d_L_d_W += self.lambda_reg * self.W

        return d_L_d_W, d_L_d_B
```

```python
class BinaryLogisticRegression(object):
    def fit(
        self,
        X: npt.ArrayLike,
        Y: npt.ArrayLike,
        X_val: npt.ArrayLike,
        Y_val: npt.ArrayLike,
        plot: bool = False,
    ):
        """
        Fit the logistic regression model to the training data.
        """
        n = X.shape[0]

        self.class_weights = compute_class_weight(
            class_weight="balanced", classes=np.array([0, 1]), y=Y
        )

        losses_train = []
        losses_val = []

        for epoch in range(self.epochs):
            epoch_loss_train = 0.0
            for i in range(0, n, self.batch_size):
                X_batch = X[i : i + self.batch_size]
                Y_batch = Y[i : i + self.batch_size]

                Y_hat = self.forward(X_batch)
                d_L_d_W, d_L_d_B = self._backward(X_batch, Y_batch, Y_hat)

                self.W -= self.learning_rate * d_L_d_W
                self.B -= self.learning_rate * d_L_d_B

                batch_loss = weighted_binary_cross_entropy_loss(
                    Y_hat, Y_batch, self.class_weights
                )
                epoch_loss_train += batch_loss * len(Y_batch)

            loss_train = epoch_loss_train / n

            Y_hat_val = self.forward(X_val)
            loss_val = weighted_binary_cross_entropy_loss(
                Y_hat_val, Y_val, self.class_weights
            )

            losses_train.append(loss_train)
            losses_val.append(loss_val)

            if plot:
                print(f"Epoch {epoch}: Train Loss: {loss_train}, Val Loss: {loss_val}")

        if plot:
            plt.plot(losses_train, label="Training Loss")
            plt.plot(losses_val, label="Validation Loss")
            plt.xlabel("Number of epochs")
            plt.ylabel("Loss")
            plt.title("Training and Validation Loss")
            plt.legend()
            plt.show()
            print("Training complete.")
```

You, 1 hour ago | 1 author (You)

```python
class BinaryNeuralNetwork(object):
    name = "Binary Neural Network"

    def __init__(
        self,
        n_features: int,
        n_hidden_units: int,
        n_hidden_layers: int,
        epochs: int = 10,
        learning_rate: float = 0.05,
        batch_size: int = 32,
        lambda_reg: float = 0.01,
    ):
        """
        X: n_input x d_input
        Y: n_input

        self.W[0]: n_features x n_hidden_units
        self.B[0]: n_hidden_units

        self.W[i: 0 < i < (n_hidden-1)]: n_hidden_units x n_hidden_units
        self.B[i: 0 < i < (n_hidden-1)]: n_hidden_units

        self.W[-1]: n_hidden_units
        self.B[-1]: 1
        """
        self.n_features = n_features
        self.n_hidden_units = n_hidden_units
        self.n_hidden_layers = n_hidden_layers
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.lambda_reg = lambda_reg

        self.class_weights = np.zeros(2)

        self.W = []
        self.B = []

        self.initialize_params()

    def initialize_params(self):
        """Initialize weights and biases (He initialization)"""
        for i in range(self.n_hidden_layers + 1):
            if i == 0:  # input -> h_1
                limit = np.sqrt(2 / (self.n_features))
                w = np.random.normal(0, limit, (self.n_features, self.n_hidden_units))
                b = np.zeros((self.n_hidden_units,))
            elif i < self.n_hidden_layers:  # h_i -> h_{i+1}
                limit = np.sqrt(2 / (self.n_hidden_units))
                w = np.random.normal(
                    0, limit, (self.n_hidden_units, self.n_hidden_units)
                )
                b = np.zeros((self.n_hidden_units,))
            else:  # h_{n} -> output
                limit = np.sqrt(2 / (self.n_hidden_units))
                w = np.random.normal(0, limit, (self.n_hidden_units, 1))
                b = np.zeros((1,))

            self.W.append(w)
            self.B.append(b)      You, last week • feat: own models
```

```python
class BinaryNeuralNetwork(object):

    def forward(self, X: npt.ArrayLike, return_intermediates=False) -> npt.ArrayLike:
        """
        Compute forward pass of the neural network.

        Y_hat: n_samples
        """
        A = [X]
        Z = []

        # forward propagation
        a_i = X
        for i in range(self.n_hidden_layers + 1):
            z_i = np.dot(a_i, self.W[i]) + self.B[i]  # add bias row-wise

            if i < self.n_hidden_layers:
                a_i = relu(z_i)
            else:  # output layer
                z_i = z_i.squeeze()  # transform (n_samples, 1) to (n_samples,)
                a_i = sigmoid(z_i)  # (n_samples,)

            Z.append(z_i)
            A.append(a_i)

        Y_hat = A[-1].squeeze()

        if return_intermediates:
            return Y_hat, A, Z

        return Y_hat

    def predict(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Create a prediction matrix with `self.forward()`

        pred: (n_samples,)
        """
        y_hat = self.forward(X)
        pred = np.where(y_hat >= 0.5, 1, 0)

        return pred

    def predict_proba(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Create a prediction matrix with `self.forward()`

        y_hat: (n_samples,)
        """
        y_hat = self.forward(X)
        return y_hat
```

You, 1 hour ago | 1 author (You)

```python
class BinaryNeuralNetwork(object):
    name = "Binary Neural Network"

    def __init__(
        self,
        n_features: int,
        n_hidden_units: int,
        n_hidden_layers: int,
        epochs: int = 10,
        learning_rate: float = 0.05,
        batch_size: int = 32,
        lambda_reg: float = 0.01,
    ):
        """
        X: n_input x d_input
        Y: n_input

        self.W[0]: n_features x n_hidden_units
        self.B[0]: n_hidden_units

        self.W[i: 0 < i < (n_hidden-1)]: n_hidden_units x n_hidden_units
        self.B[i: 0 < i < (n_hidden-1)]: n_hidden_units

        self.W[-1]: n_hidden_units
        self.B[-1]: 1
        """
        self.n_features = n_features
        self.n_hidden_units = n_hidden_units
        self.n_hidden_layers = n_hidden_layers
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.lambda_reg = lambda_reg

        self.class_weights = np.zeros(2)

        self.W = []
        self.B = []

        self.initialize_params()

    def initialize_params(self):
        """Initialize weights and biases (He initialization)"""
        for i in range(self.n_hidden_layers + 1):
            if i == 0:  # input -> h_1
                limit = np.sqrt(2 / (self.n_features))
                w = np.random.normal(0, limit, (self.n_features, self.n_hidden_units))
                b = np.zeros((self.n_hidden_units,))
            elif i < self.n_hidden_layers:  # h_i -> h_{i+1}
                limit = np.sqrt(2 / (self.n_hidden_units))
                w = np.random.normal(
                    0, limit, (self.n_hidden_units, self.n_hidden_units)
                )
                b = np.zeros((self.n_hidden_units,))
            else:  # h_{n} -> output
                limit = np.sqrt(2 / (self.n_hidden_units))
                w = np.random.normal(0, limit, (self.n_hidden_units, 1))
                b = np.zeros((1,))

            self.W.append(w)
            self.B.append(b)      You, last week • feat: own models
```

```python
class BinaryNeuralNetwork(object):

    def forward(self, X: npt.ArrayLike, return_intermediates=False) -> npt.ArrayLike:
        """
        Compute forward pass of the neural network.

        Y_hat: n_samples
        """
        A = [X]
        Z = []

        # forward propagation
        a_i = X
        for i in range(self.n_hidden_layers + 1):
            z_i = np.dot(a_i, self.W[i]) + self.B[i]  # add bias row-wise

            if i < self.n_hidden_layers:
                a_i = relu(z_i)
            else:  # output layer
                z_i = z_i.squeeze()  # transform (n_samples, 1) to (n_samples,)
                a_i = sigmoid(z_i)  # (n_samples,)

            Z.append(z_i)
            A.append(a_i)

        Y_hat = A[-1].squeeze()

        if return_intermediates:
            return Y_hat, A, Z

        return Y_hat

    def predict(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Create a prediction matrix with `self.forward()`

        pred: (n_samples,)
        """
        y_hat = self.forward(X)
        pred = np.where(y_hat >= 0.5, 1, 0)

        return pred

    def predict_proba(self, X: npt.ArrayLike) -> npt.ArrayLike:
        """
        Create a prediction matrix with `self.forward()`

        y_hat: (n_samples,)
        """
        y_hat = self.forward(X)
        return y_hat
```

```python
class BinaryNeuralNetwork(object):

    def _backward(
        self,
        Y: npt.ArrayLike,
        Y_hat: npt.ArrayLike,
        A: npt.ArrayLike,
        Z: npt.ArrayLike,
    ) -> Tuple[npt.ArrayLike, npt.ArrayLike]:
        """
        Compute back propagation of the neural network.
        """
        batch_size = Y.shape[0]

        d_L_d_W = [np.zeros_like(w) for w in self.W]
        d_L_d_B = [np.zeros_like(b) for b in self.B]

        # derivative of the loss with respect to the pre-activation of the output layer
        d_L_d_z = delta_weighted_bce(Y_hat, Y, self.class_weights)
        d_L_d_z = d_L_d_z.reshape(-1, 1)  # reshape from (n_samples,) to (n_samples, 1)

        # derivatives of the loss with respect to the weights and biases of the last layer
        d_L_d_W[-1] = np.dot(A[-2].T, d_L_d_z) / batch_size
        d_L_d_B[-1] = np.sum(d_L_d_z, axis=0) / batch_size

        # back propagation using chain rule
        for l in range(self.n_hidden_layers - 1, -1, -1):
            d_L_d_z = np.dot(d_L_d_z, self.W[l + 1].T) * relu_prime(
                Z[l]
            )  # d^l = (d^{l+1})^T * W^{l+1} x (dA^{l} / dz^{l})

            d_L_d_W[l] = (
                np.dot(A[l].T, d_L_d_z) / batch_size
            )  # note: A[l] is actually correct here - A[0] is X!
            d_L_d_B[l] = np.sum(d_L_d_z, axis=0) / batch_size

            # add L2 regularization derivative
            d_L_d_W[l] += self.lambda_reg * self.W[l]

        return d_L_d_W, d_L_d_B
```

```python
class BinaryNeuralNetwork(object):
    def fit(
        self,
        X: npt.ArrayLike,
        Y: npt.ArrayLike,
        X_val: npt.ArrayLike,
        Y_val: npt.ArrayLike,
        plot: bool = False,
    ):
        """
        Fit the neural network to the training data.
        """
        n = X.shape[0]
        self.class_weights = compute_class_weight(
            class_weight="balanced", classes=np.array([0, 1]), y=Y
        )

        losses_train = []
        losses_val = []

        for epoch in range(self.epochs):
            epoch_loss_train = 0.0
            for i in range(0, n, self.batch_size):
                X_batch = X[i : i + self.batch_size]
                Y_batch = Y[i : i + self.batch_size]

                Y_hat, A, Z = self.forward(X_batch, return_intermediates=True)
                d_L_d_W, d_L_d_B = self._backward(Y_batch, Y_hat, A, Z)

                for j in range(self.n_hidden_layers + 1):
                    self.W[j] -= self.learning_rate * d_L_d_W[j]
                    self.B[j] -= self.learning_rate * d_L_d_B[j]

                batch_loss = weighted_binary_cross_entropy_loss(
                    Y_hat, Y_batch, self.class_weights
                )
                epoch_loss_train += batch_loss * len(Y_batch)

            loss_train = epoch_loss_train / n

            Y_hat_val = self.forward(X_val)
            loss_val = weighted_binary_cross_entropy_loss(
                Y_hat_val, Y_val, self.class_weights
            )

            losses_train.append(loss_train)
            losses_val.append(loss_val)

            if plot:
                print(f"Epoch {epoch}: Train Loss: {loss_train}, Val Loss: {loss_val}")

        if plot:
            plt.plot(losses_train, label="Training Loss")
            plt.plot(losses_val, label="Validation Loss")
            plt.xlabel("Number of epochs")
            plt.ylabel("Loss")
            plt.title("Training and Validation Loss")
            plt.legend()
            plt.show()
            print("Training complete.")
```

```python
class BinaryNeuralNetwork(object):

    def _backward(
        self,
        Y: npt.ArrayLike,
        Y_hat: npt.ArrayLike,
        A: npt.ArrayLike,
        Z: npt.ArrayLike,
    ) -> Tuple[npt.ArrayLike, npt.ArrayLike]:
        """
        Compute back propagation of the neural network.
        """
        batch_size = Y.shape[0]

        d_L_d_W = [np.zeros_like(w) for w in self.W]
        d_L_d_B = [np.zeros_like(b) for b in self.B]

        # derivative of the loss with respect to the pre-activation of the output layer
        d_L_d_z = delta_weighted_bce(Y_hat, Y, self.class_weights)
        d_L_d_z = d_L_d_z.reshape(-1, 1)  # reshape from (n_samples,) to (n_samples, 1)

        # derivatives of the loss with respect to the weights and biases of the last layer
        d_L_d_W[-1] = np.dot(A[-2].T, d_L_d_z) / batch_size
        d_L_d_B[-1] = np.sum(d_L_d_z, axis=0) / batch_size

        # back propagation using chain rule
        for l in range(self.n_hidden_layers - 1, -1, -1):
            d_L_d_z = np.dot(d_L_d_z, self.W[l + 1].T) * relu_prime(
                Z[l]
            )  # d^l = (d^{l+1})^T * W^{l+1} x (dA^{l} / dz^{l})

            d_L_d_W[l] = (
                np.dot(A[l].T, d_L_d_z) / batch_size
            )  # note: A[l] is actually correct here - A[0] is X!
            d_L_d_B[l] = np.sum(d_L_d_z, axis=0) / batch_size

            # add L2 regularization derivative
            d_L_d_W[l] += self.lambda_reg * self.W[l]

        return d_L_d_W, d_L_d_B
```

```python
class BinaryNeuralNetwork(object):
    def fit(
        self,
        X: npt.ArrayLike,
        Y: npt.ArrayLike,
        X_val: npt.ArrayLike,
        Y_val: npt.ArrayLike,
        plot: bool = False,
    ):
        """
        Fit the neural network to the training data.
        """
        n = X.shape[0]
        self.class_weights = compute_class_weight(
            class_weight="balanced", classes=np.array([0, 1]), y=Y
        )

        losses_train = []
        losses_val = []

        for epoch in range(self.epochs):
            epoch_loss_train = 0.0
            for i in range(0, n, self.batch_size):
                X_batch = X[i : i + self.batch_size]
                Y_batch = Y[i : i + self.batch_size]

                Y_hat, A, Z = self.forward(X_batch, return_intermediates=True)
                d_L_d_W, d_L_d_B = self._backward(Y_batch, Y_hat, A, Z)

                for j in range(self.n_hidden_layers + 1):
                    self.W[j] -= self.learning_rate * d_L_d_W[j]
                    self.B[j] -= self.learning_rate * d_L_d_B[j]

                batch_loss = weighted_binary_cross_entropy_loss(
                    Y_hat, Y_batch, self.class_weights
                )
                epoch_loss_train += batch_loss * len(Y_batch)

            loss_train = epoch_loss_train / n

            Y_hat_val = self.forward(X_val)
            loss_val = weighted_binary_cross_entropy_loss(
                Y_hat_val, Y_val, self.class_weights
            )

            losses_train.append(loss_train)
            losses_val.append(loss_val)

            if plot:
                print(f"Epoch {epoch}: Train Loss: {loss_train}, Val Loss: {loss_val}")

        if plot:
            plt.plot(losses_train, label="Training Loss")
            plt.plot(losses_val, label="Validation Loss")
            plt.xlabel("Number of epochs")
            plt.ylabel("Loss")
            plt.title("Training and Validation Loss")
            plt.legend()
            plt.show()
            print("Training complete.")
```

```python
def weighted_binary_cross_entropy_loss(
    output, target, class_weights=None, epsilon=0.001
):
    """
    Compute the weighted binary cross-entropy loss.
    """
    if class_weights is None:
        class_weights = compute_class_weights(target)

    # avoiding numerical instability
    output = np.clip(output, epsilon, 1 - epsilon)

    loss = class_weights[1] * (target * np.log(output)) + class_weights[0] * (
        (1 - target) * np.log(1 - output)
    )

    return -np.mean(loss)


def compute_class_weights(y: npt.ArrayLike) -> npt.ArrayLike:
    """
    Compute class weights for imbalanced datasets.
    """
    y = np.array(y, dtype=np.int64)
    classes = np.unique(y)
    class_counts = np.bincount(y)
    total_samples = len(y)

    class_weights = {
        cls: total_samples / (len(classes) * count)
        for cls, count in zip(classes, class_counts)
    }

    weights_array = np.array([class_weights[cls] for cls in classes], dtype=np.float64)

    return weights_array
```

```python
def weighted_binary_cross_entropy_loss(
    output, target, class_weights=None, epsilon=0.001
):
    """
    Compute the weighted binary cross-entropy loss.
    """
    if class_weights is None:
        class_weights = compute_class_weights(target)

    # avoiding numerical instability
    output = np.clip(output, epsilon, 1 - epsilon)

    loss = class_weights[1] * (target * np.log(output)) + class_weights[0] * (
        (1 - target) * np.log(1 - output)
    )

    return -np.mean(loss)


def compute_class_weights(y: npt.ArrayLike) -> npt.ArrayLike:
    """
    Compute class weights for imbalanced datasets.
    """
    y = np.array(y, dtype=np.int64)
    classes = np.unique(y)
    class_counts = np.bincount(y)
    total_samples = len(y)

    class_weights = {
        cls: total_samples / (len(classes) * count)
        for cls, count in zip(classes, class_counts)
    }

    weights_array = np.array([class_weights[cls] for cls in classes], dtype=np.float64)

    return weights_array
```

4. Feature Selection for Logistic Regression

- Polynomial Features, Primal Decision Function
- Principal Components through Principal Component Analysis
- Feature Selection using custom backward selection protocol

4. Feature Selection for Logistic Regression

- Feature Selection Protocol: Backward Selection
  - delete one feature per iteration
  - train LogisticRegression model, calculate risk estimate on validation set at each step
  - as soon as all but one features are deleted: return the features that the model with the lowest risk estimate was trained on

```python
def feature_selection(
    X,
    y,
    n_features_per_iteration: int = 1,
    plot: bool = False,
):
    n_features = X.shape[1]
    deleted_features = []
    optimal_number_of_features_to_delete = 0

    risk_estimates = [np.inf]
    lowest_risk_estimate = np.inf

    X_tmp = X.copy()
    y_tmp = y.copy()

    original_indices = np.arange(n_features)

    for i in range(0, n_features - 1):
        X_train, X_val, y_train, y_val = train_test_split(
            X_tmp, y_tmp, test_size=0.2, random_state=42
        )
        model = LogisticRegression(
            penalty="l1",
            solver="liblinear",
            max_iter=1000,
            class_weight="balanced",
        )
        log_reg_tmp = model.fit(X_train, y_train)
        y_pred = log_reg_tmp.predict_proba(X_val)[:, 1]
        risk_estimate = weighted_binary_cross_entropy_loss(y_pred, y_val)
        log_message = f"i = {i} / {n_features-1} - Empirical Risk Estimate: {risk_estimate}, Previous Estimate: {risk_estimates[-1]}"

        features_to_delete = np.argsort(np.abs(log_reg_tmp.coef_))[0][
            :n_features_per_iteration
        ]
        deleted_features.extend(original_indices[features_to_delete].tolist())

        X_tmp = np.delete(X_tmp, features_to_delete, axis=1)
        original_indices = np.delete(original_indices, features_to_delete)

        risk_estimates.append(risk_estimate)

        if risk_estimate < lowest_risk_estimate:
            lowest_risk_estimate = risk_estimate
            optimal_number_of_features_to_delete = i
            log_message += " - new lowest risk estimate"

        print(log_message)

    features_to_delete = deleted_features[:optimal_number_of_features_to_delete]

    if plot:
        print(
            f"Number of features to delete: {optimal_number_of_features_to_delete} / {n_features}"
        )

        plt.plot(
            range(
                0, n_features - 1 + n_features_per_iteration, n_features_per_iteration
            ),
            risk_estimates,
            label="Training Loss",
        )
        plt.xlabel("Number of deleted features")
        plt.ylabel("Weighted Binary Cross Entropy Loss")
        plt.title("Risk estimate as a function of deleted features")
        plt.legend()
        plt.show()

    return features_to_delete, risk_estimates
```

5. Model Selection & Evaluation Protocol

- Nested Cross-Validation protocol for hyper parameter tuning and evaluation
  - 5-fold outer loop, 5-fold inner loop
    - inner loop: iterate over all given hyper parameter sets, calculate empirical risk estimate per iteration using 5-fold cross-validation
    - outer loop: 5-fold split of the data
  - after outer loop: average empirical risk estimates per hyper parameter set, pick set with lowest empirical risk estimate, evaluate model using 5-fold cross-validation, fit final model on whole training set

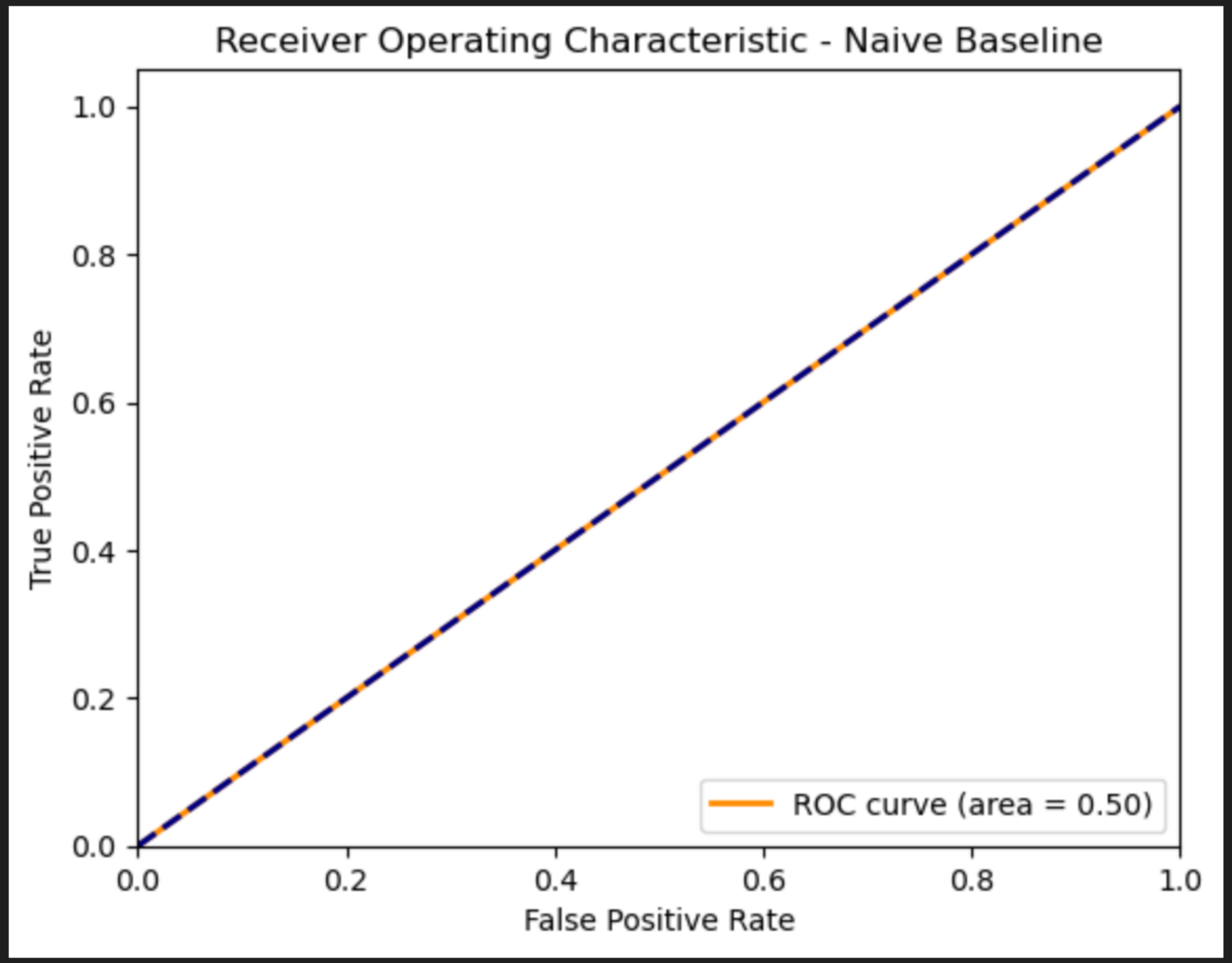
```
param_grid_lr = {
    "n_features": [X_selected.shape[1]],
    "epochs": [10, 20, 30],
    "learning_rate": [0.1, 0.05, 0.01],
    "lambda_reg": [0.1, 0.01, 0.001],
}
```

```
param_grid_nn = {
    "n_features": [X_original.shape[1]],
    "epochs": [10, 20, 30],
    "learning_rate": [0.1, 0.05, 0.01],
    "lambda_reg": [0.1, 0.01, 0.001],
    "n_hidden_units": [16, 32, 64],
    "n_hidden_layers": [1, 2],
}
```

```python
def nested_cross_validation(X, y, Model, param_grid, k=5):
    print(
        f"Performing nested cross-validation for model {Model.name} with {X.shape[0]} samples"
    )
    # list of one dict per parameter combination
    param_combinations = [
        dict(zip(param_grid.keys(), values)) for values in product(*param_grid.values())
    ]
    n_combinations = len(param_combinations)

    R_ests = np.zeros((k, len(param_combinations)))

    kf_outer = KFold(n_splits=k, shuffle=True)

    for i, (outer_train_idx, outer_test_idx) in enumerate(kf_outer.split(X)):
        X_train_outer, X_val_outer = X[outer_train_idx], X[outer_test_idx]
        y_train_outer, y_val_outer = y[outer_train_idx], y[outer_test_idx]

        for j, params in enumerate(param_combinations):
            model = Model(**params)

            R_ests[i, j] = k_fold_cross_validation(
                model, X_train_outer, y_train_outer, k=k, fit_final_model=False
            )
            print(
                f"Empirical risk estimate for fold {i+1} / {k}, parameter set {j+1} / {n_combinations}: {R_ests[i, j]}",
                end="\r",
            )

    R_ests_params = np.mean(R_ests, axis=0)

    params = param_combinations[np.argmin(R_ests_params)]
    print(
        f"Risk estimate for argmin(R_ests_params): {R_ests_params[np.argmin(R_ests_params)]}"
    )
    print(f"Selected best parameters: {params}")

    model_final = Model(**params)
    model_final_trained, R_est, acc, f1, auc, prec, rec, cm_df, fpr, tpr = (
        k_fold_cross_validation(model_final, X, y, k=k, fit_final_model=True)
    )
    print(
        f"{Model.name} - Empirical Risk Estimate: {R_est}, Accuracy: {acc}, F1 score: {f1}, Precision: {prec}, Recall: {rec}, AUC: {auc}\n{cm_df}"
    )

    # Plot ROC curve
    plt.figure()
    plt.plot(fpr, tpr, color="darkorange", lw=2, label=f"ROC curve (area = {auc:0.2f})")
    plt.plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title(f"Receiver Operating Characteristic - {Model.name}")
    plt.legend(loc="lower right")
    plt.show()

    return model_final_trained, params, R_est, acc, f1, auc
```

6. Results – Naive Baseline

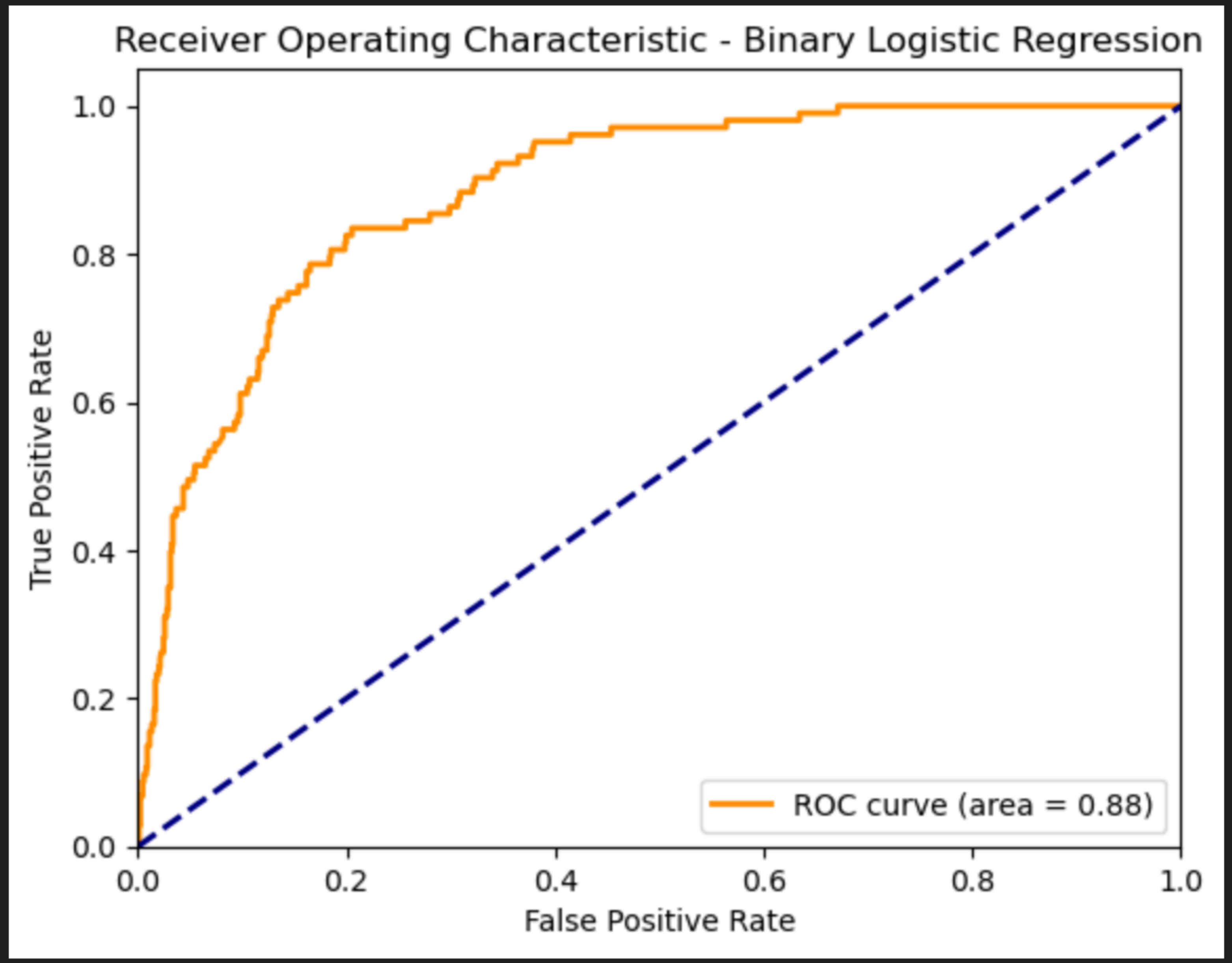
```
2024-08-04 17:45:48.570 | INFO     | src.training.evaluate:evaluate:40 - Naive Baseline - Empirical Risk Estimate: 1.211828993715628, Accuracy: 0.797, F1 score: 0.01932367149758454, Precision: 0.0196078431372549, Recall: 0.01904761904761905, AUC: 0.5
            Pred Neg  Pred Pos
Actual Neg       795       100
Actual Pos       103         2
Validation accuracy: 0.835
```



6. Results – Logistic Regression

[49] ✓ 2m 5.4s

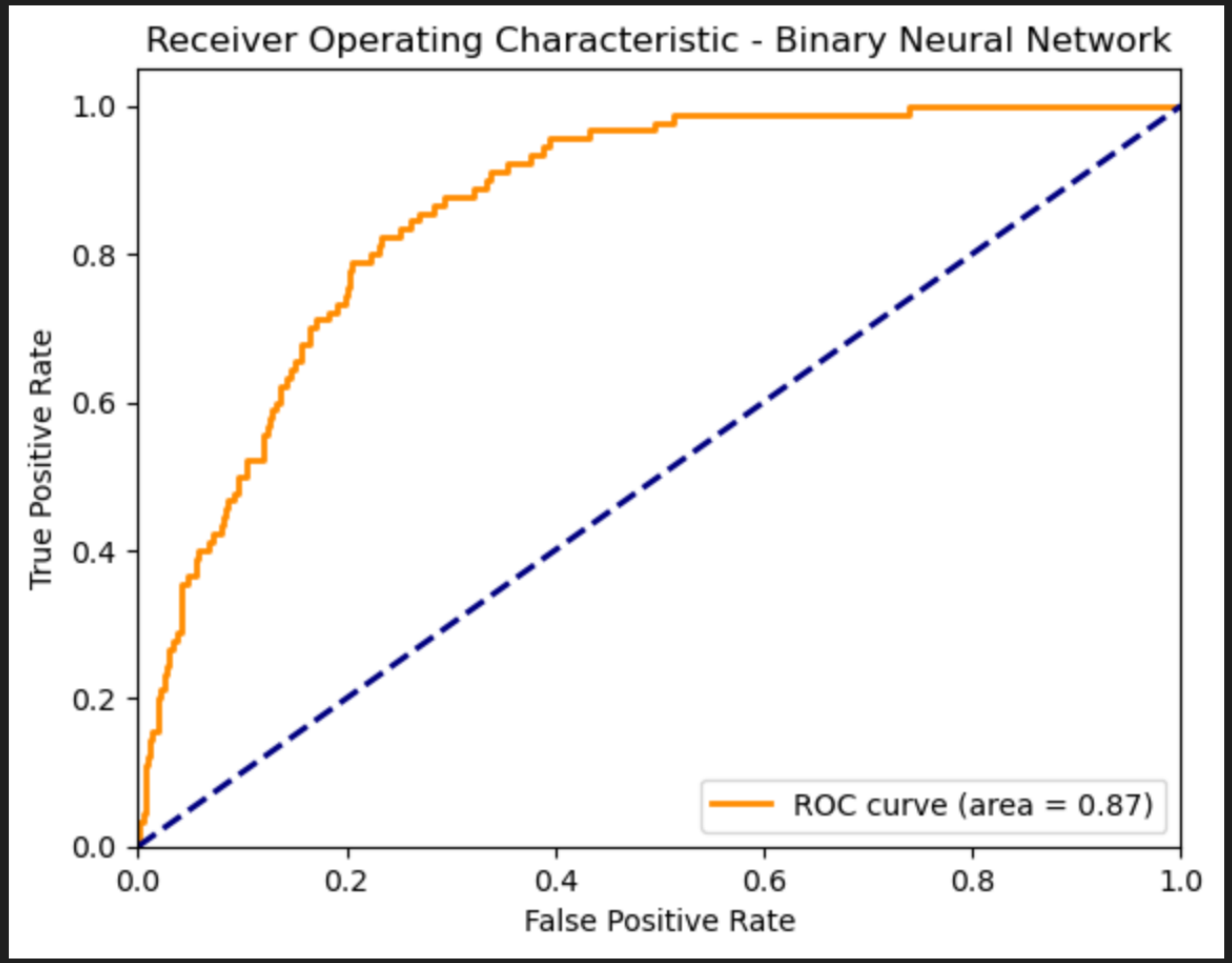
```
Performing nested cross-validation for model Binary Logistic Regression with 5000 samples
Risk estimate for argmin(R_ests_params): 0.431203867143375447: 0.44956165679782895
Selected best parameters: {'n_features': 3, 'epochs': 20, 'learning_rate': 0.1, 'lambda_reg': 0.001}
Binary Logistic Regression – Empirical Risk Estimate: 0.4326625244737709, Accuracy: 0.7598, F1 score: 0.4106326848282034, Precision: 0.27178922773035485, Recall: 0.8410820460604125, AUC: 0.8783259164403588
            Pred Neg  Pred Pos
Actual Neg       678       219
Actual Pos        17        86
```



6. Results – Neural Network

[50] 20m 28.7s

```
Performing nested cross-validation for model Binary Neural Network with 5000 samples
Risk estimate for argmin(R_ests_params): 0.43757705619196885162: 0.45009287877300624
Selected best parameters: {'n_features': 16, 'epochs': 30, 'learning_rate': 0.05, 'lambda_reg': 0.01, 'n_hidden_units': 32, 'n_hidden_layers': 1}
Binary Neural Network – Empirical Risk Estimate: 0.43897262965985584, Accuracy: 0.7506, F1 score: 0.4066613515367429, Precision: 0.2672795367251504, Recall: 0.85771651065744, AUC: 0.8732469714613273
            Pred Neg  Pred Pos
Actual Neg       655       255
Actual Pos        13        77
```



6. Results – Comparison

- Logistic Regression model slightly outperforms Neural Network model regarding AUC and F1
  - Logistic Regression model is better at separating classes across decision thresholds and at threshold of 0.5
- Neural Network model slightly outperforms Logistic Regression model in recall, but lacks behind in precision
  - It is more liberal: less false negatives, but more false positives

7. Conclusion

- The Logistic Regression model slightly outperforms the Neural Network
  - AUC metric is especially important in medical context
  - Minimizing false negatives may be more important than maximizing F1
- The competitive edge of the Logistic Regression model is not large
- Neural Network’s ability to derive non-linear relations on its own does not grant performance edge, given the data set at hand

7. Conclusion

- Future work could focus on:
  - Leveraging more comprehensive data set and more complex Neural Network architecture: Might boost Neural Network performance
  - Creating higher-order polynomials and use dual decision functions: Might boost Logistic Regression performance and computational efficiency – but might as well lead to overfitting
  - Implementing adaptive learning in Neural Network: potentially allow convergence in better minima