Predicting Left Ventricular Ejection Fraction Using Artificial Intelligence (AI) to Save Costs by Using Electrocardiograms (ECGs) over Echocardiograms.

Predicting left ventricular ejection fraction using artificial intelligence (AI) to save costs by using electrocardiograms (ECGs) over echocardiograms.

Abstract

Left ventricular ejection fraction (LVEF), also known as ejection fraction (EF), is an abstract measure of your heart's ability to pump blood - specifically from the left ventricle, or the lower left part of the heart which sends blood gushing through the aorta into the rest of the body for several purposes (oxygenation, hydration, etc.). However, as one's heart gets weaker, the body's ability to conduct several functions gets hindered, creating a variety of health problems. That is where echocardiograms, "echo" for short, comes into play as they record your heart and get examined to deduce the patient's EF. There is a massive downside, however, which is that echocardiograms are highly expensive, preventing or deterring most people from taking it. According to a study by AskChapter.org (7), echocardiograms may cost up to $3000, mostly unaffordable - especially for those who need it the most. That is why most people prefer an electrocardiogram (Ecg or ekg) to track bodily health. According to mdsave.com (8), Ecgs can cost as low as $50 - way more affordable. Unfortunately however, an Ecg only takes certain electrical measurements and doesn't exactly measure EF. That is where this LVEF-prediction artificially intelligent network comes into the scene, which when you input your age, gender, race, and other electrocardiographic parameters (specifically qrs-duration, qt-duration, and ventricular rate), it outputs the LVEF. This project aims to incorporate both the analytical aspects of cardiovascular dysfunction pertaining to where and how all six parameters affect EF, and the neural network aspect which entail how well the AI does and where it goes wrong.

Hypothesis

I believe this AI will be clinically accurate to predict EF or more specifically LVEF because it had been trained off a reliable and large-sized biorepository from the Massachusetts Institute of Technology (MIT) and used computationally intensive yet efficient backpropagation and gradient descent algorithm customized by its creator which would be myself. I also believe my AI to be accurate on all circumstances related

to electrocardiography - which would be qrs-duration, qt-duration, and ventricular rate - as they are continuous and are measured with only marginal error, but will be bad at recognizing patterns with age, gender, and race as they are discrete, non-numerical, and or irrelevant in some circumstances. On the analytical side of things, older individuals may have a lower EF as their hearts begin to weaken after the age of 65 or so (4). Females may have a higher ejection fraction than men due to their natural physical longevity. For race, a truly backed-up hypothesis may not be backed up because of its generality and non-specificity, but the results may tell a different story. Finally, for qrs-duration, qt-duration, and ventricular rate, the higher values indicate a longer period in the heart's blood-ejection, indicating a lower left ventricular ejection fraction.

Materials and Procedures

Materials:

To conduct this experiment, you would have to acquire a computer with a cpu, the following programming (listed in this section), and some form of Ecg data (collected or acquired).

Data collection:

I first found my database - Echonext_metadata_100k_samples made by Pierre Elias and Joshua Fisher (9) - on Physionet, one of the top medical database repositories. This database contains 100,000 samples from people in Boston from all backgrounds. The database contains information about age, gender, race, and all other categories with minimal bias.

Prototype:

In my attempt to make an AI, I started with a crude model which had 500 lines of code and was suboptimal. My cpu setup with an altered cache was inefficient and would've taken 10 days to complete training and several hours to fetch a result with less than 10% accuracy. This is why I changed my model drastically with a new efficient cache system and a much better optimizer. It is also only 250 lines of code.

Code:

This was my first attempt at making the AI:

```
import numpy as np
```

```python
import pandas as pd

'''Attributes listed here:'''
# df.iloc[row, column] is how you retrieve a specific number
# df[name] is how you get a specific column
# df.loc[row, name] is another way of retrieving a specific number
# np.ones((rows, columns)) is how to get a 2d array
# np.array[] makes a 1d array
# X.T is a transposed matrix
# np.dot is a dot product


'''Load in Spreadsheet'''
df = pd.read_csv("echonext_metadata_100k.csv")

'''I/O vectors'''
age = []
sex = []
race = []
qrs_duration = []
qt_corrected = []
ventricular_rate = []
lvef_value = []

a, b, c, d, e, f, g = 0, 0, 0, 0, 0, 0, 0

# If this doesn't work its outdated and replace with df.loc
for a in range(80000):
   age.append(df.iloc[a]["age_at_ecg"])
for b in range(80000):
   sex.append(df.iloc[b]["sex"])
for c in range(80000):
   race.append(df.iloc[c]["race_ethnicity"])
for d in range(80000):
   qrs_duration.append(df.iloc[d]["qrs_duration"])
for e in range(80000):
   qt_corrected.append(df.iloc[e]["qt_corrected"])
for f in range(80000):
   ventricular_rate.append(df.iloc[f]["ventricular_rate"])
for g in range(80000):
   lvef_value.append(df.iloc[g]["lvef_value"])

# Convert numeric columns to numpy arrays
age = np.array(age, dtype=float)
qrs_duration = np.array(qrs_duration, dtype=float)
qt_corrected = np.array(qt_corrected, dtype=float)
ventricular_rate = np.array(ventricular_rate, dtype=float)
lvef_value = np.array(lvef_value, dtype=float)
```

```
# Encode sex: Male = 1, Female = 0, Unknown/Other = 0
sex_numeric = []
for s in sex:
   s_str = str(s).strip().lower()
   if s_str.startswith("m"):
       sex_numeric.append(1)
   elif s_str.startswith("f"):
       sex_numeric.append(0)
   else:
       sex_numeric.append(0)
sex_numeric = np.array(sex_numeric, dtype=float)

# Encode race:
# black = 1, white = 2, hispanic = 3, asian = 4, other = 5, unknown = 0
race_numeric = []
for r in race:
   r_str = str(r).strip().lower()
   if "black" in r_str:
       race_numeric.append(1)
   elif "white" in r_str:
       race_numeric.append(2)
   elif "hispanic" in r_str:
       race_numeric.append(3)
   elif "asian" in r_str:
       race_numeric.append(4)
   elif "other" in r_str:
       race_numeric.append(5)
   elif "unknown" in r_str or r_str in ["nan", "none", ""]:
       race_numeric.append(0)
   else:
       race_numeric.append(0)
race_numeric = np.array(race_numeric, dtype=float)

'''Weights and biases'''


class Weights:
   def __init__(self):
       self.wlayer_one = np.ones((10, 6), dtype=float)
       self.wlayer_two = np.ones((10, 10), dtype=float)
       self.wlayer_three = np.ones((10, 10), dtype=float)
       self.wlayer_four = np.ones((10, 10), dtype=float)
       self.wlayer_five = np.ones((10, 10), dtype=float)
       self.wlayer_six = np.ones((10, 10), dtype=float)
       self.wlayer_seven = np.ones((10, 10), dtype=float)
       self.wlayer_eight = np.ones((10, 10), dtype=float)
       self.wlayer_nine = np.ones((10, 10), dtype=float)
       self.wlayer_ten = np.ones((10, 10), dtype=float)
```

```
        self.wlayer_final = np.ones((1, 10), dtype=float)  # make 2D for batch
compat


class Biases:
    def __init__(self):
        self.blayer_one = np.ones(10, dtype=float)
        self.blayer_two = np.ones(10, dtype=float)
        self.blayer_three = np.ones(10, dtype=float)
        self.blayer_four = np.ones(10, dtype=float)
        self.blayer_five = np.ones(10, dtype=float)
        self.blayer_six = np.ones(10, dtype=float)
        self.blayer_seven = np.ones(10, dtype=float)
        self.blayer_eight = np.ones(10, dtype=float)
        self.blayer_nine = np.ones(10, dtype=float)
        self.blayer_ten = np.ones(10, dtype=float)
        self.blayer_final = np.array([1.0], dtype=float)  # single output


Biases = Biases()
Weights = Weights()

'''Input Activation'''


def relu(x):
    return np.maximum(0, x)


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


'''Martix multiplier'''


def multiply(x, y):
    return np.dot(x.T, y)


'''Learning Rate'''
ln = 0.01
eps = 0.000001

'''Loss and Cost'''


def loss(x, y):
    return np.square(x - y).mean()
```

```
def cost(x, y, z):
   return z * sigmoid(loss(x, y))


'''Gradient Descent'''

# x are weights and biases
# y is the previous neuron
# z is neuron expected
def newton_optimizer(x, y_batch, z_batch):
   x = x.copy()
   batch_size = y_batch.shape[0]

   if x.ndim == 1:
       for i in range(x.shape[0]):
           update_sum = 0.0
           for batch in range(batch_size):
               y = y_batch[batch, i] if y_batch.ndim > 1 else y_batch[batch]
               z = z_batch[batch] if z_batch.ndim > 1 else z_batch[batch]

               f0 = cost(y * x[i], z, y)
               f1 = cost(y * (x[i] + eps), z, y)
               f2 = cost(y * (x[i] - eps), z, y)

               f_prime = (f1 - f2) / (2 * eps)
               f_double_prime = (f1 - 2 * f0 + f2) / (eps ** 2)

               if not np.isnan(f_double_prime) and f_double_prime != 0:
                   update_sum += f_prime / f_double_prime

           # Average update over batch
           update = update_sum / batch_size
           update = np.clip(update, -eps, eps)
           x[i] -= update


   else:

       for i in range(x.shape[0]):
           for j in range(x.shape[1]):
               update_sum = 0.0
               for batch in range(batch_size):
                   # Always pick scalar from column vectors
                   y = y_batch[batch, 0] if y_batch.ndim > 1 else
y_batch[batch]
                   z = z_batch[batch, 0] if z_batch.ndim > 1 else
z_batch[batch]
```

```
                    f0 = cost(y * x[i, j], z, y)
                    f1 = cost(y * (x[i, j] + eps), z, y)
                    f2 = cost(y * (x[i, j] - eps), z, y)
                    f_prime = (f1 - f2) / (2 * eps)
                    f_double_prime = (f1 - 2 * f0 + f2) / (eps ** 2)

                    if not np.isnan(f_double_prime) and f_double_prime != 0:
                        update_sum += f_prime / f_double_prime

                update = update_sum / batch_size
                update = np.clip(update, -eps, eps)
                x[i, j] -= update
    return x


'''Final Output'''

def forward_and_train(X, Y, weights, biases, epochs=10, lr=0.01,
batch_size=32):
    n_samples = X.shape[0]

    for epoch in range(epochs):
        # Shuffle data at the start of each epoch
        indices = np.arange(n_samples)
        np.random.shuffle(indices)
        X_shuffled = X[indices]
        Y_shuffled = Y[indices]

        for start in range(0, n_samples, batch_size):
            end = start + batch_size
            X_batch = X_shuffled[start:end]
            Y_batch = Y_shuffled[start:end]

            # Forward propagation for this batch
            h1 = relu(np.dot(X_batch, weights.wlayer_one.T) +
biases.blayer_one)
            h2 = relu(np.dot(h1, weights.wlayer_two.T) + biases.blayer_two)
            h3 = relu(np.dot(h2, weights.wlayer_three.T) +
biases.blayer_three)
            h4 = relu(np.dot(h3, weights.wlayer_four.T) + biases.blayer_four)
            h5 = relu(np.dot(h4, weights.wlayer_five.T) + biases.blayer_five)
            h6 = relu(np.dot(h5, weights.wlayer_six.T) + biases.blayer_six)
            h7 = relu(np.dot(h6, weights.wlayer_seven.T) +
biases.blayer_seven)
            h8 = relu(np.dot(h7, weights.wlayer_eight.T) +
biases.blayer_eight)
            h9 = relu(np.dot(h8, weights.wlayer_nine.T) + biases.blayer_nine)
            h10 = relu(np.dot(h9, weights.wlayer_ten.T) + biases.blayer_ten)
```

```
            output = sigmoid(np.dot(h10, weights.wlayer_final.T) +
biases.blayer_final)

            # Loss for batch
            loss_val = np.square(output - Y_batch).mean()

            # Gradient updates (Newton optimizer and bias update)
            weights.wlayer_final = newton_optimizer(weights.wlayer_final,
output, Y_batch)
            biases.blayer_final -= lr * float(np.mean(output - Y_batch))
            weights.wlayer_ten = newton_optimizer(weights.wlayer_ten, h10,
Y_batch)
            weights.wlayer_nine = newton_optimizer(weights.wlayer_nine, h9,
Y_batch)
            weights.wlayer_one = newton_optimizer(weights.wlayer_one, X_batch,
Y_batch)

        print(f"Epoch {epoch + 1}/{epochs} | Loss: {loss_val:.6f}")

    return weights, biases

'''Prepare sample input/output for training'''
X = np.column_stack((age, sex_numeric, race_numeric, qrs_duration,
qt_corrected, ventricular_rate))
Y = np.array(lvef_value).reshape(-1, 1)

# Normalize (optional for better stability)
X = (X - np.mean(X, axis=0)) / (np.std(X, axis=0) + 1e-9)
Y = (Y - np.mean(Y)) / (np.std(Y) + 1e-9)

'''Train the model 1000 times'''
final_weights, final_biases = forward_and_train(X, Y, Weights, Biases,
epochs=50, lr=0.01, batch_size=32)

import pickle

# Save weights and biases to a pickle file
with open("trained_model.pkl", "wb") as f:
    pickle.dump({
        "weights": weights,
        "biases": biases
    }, f)

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
```

```
    return "Hello, Flask!"

if __name__ == "__main__":
    app.run(debug=True)

print("Trained model saved to 'trained_model.pkl'")
```

This evaluation was abysmal, with several redundancies and an unsuccessful cache system which further delayed the system processing. The Newton evaluation was overturned as it took forever to update gradients because of previous inefficiencies.

This was my second, successful attempt:

```
#  FAST AI TRAINER — 10 Hidden Layers, Newton-Like Optimizer (L-BFGS)

import pandas as pd
import numpy as np
from scipy.optimize import minimize

# ------------------------------------------------------------
# 1. Load and preprocess data
# ------------------------------------------------------------
print("Loading echonext_metadata_100k.csv ...")
df = pd.read_csv("echonext_metadata_100k.csv")

cols = [
    "age_at_ecg",
    "sex",
    "race_ethnicity",
    "qrs_duration",
    "qt_corrected",
    "ventricular_rate",
    "lvef_value"
]
df = df[cols].dropna()

print(f"Loaded {len(df)} samples after dropping NaNs.")

# Encode sex: male=1, female=0
df["sex"] = df["sex"].astype(str).str.lower().map(lambda s: 1 if
s.startswith("m") else 0)

# Encode race: text -> numeric
def encode_race(r):
    r = str(r).strip().lower()
    if "black" in r: return 1
    if "white" in r: return 2
    if "hispanic" in r: return 3
    if "asian" in r: return 4
    if "other" in r: return 5
    return 0

df["race_ethnicity"] = df["race_ethnicity"].apply(encode_race)
```

```
# Inputs and outputs
X = df[["age_at_ecg", "sex", "race_ethnicity", "qrs_duration",
"qt_corrected", "ventricular_rate"]].values
Y = df["lvef_value"].values.reshape(-1, 1)

# Normalize
X_mean, X_std = np.mean(X, axis=0), np.std(X, axis=0) + 1e-9
Y_mean, Y_std = np.mean(Y), np.std(Y) + 1e-9
X = (X - X_mean) / X_std
Y = (Y - Y_mean) / Y_std

# ------------------------------------------------------------
# 2. Define network (10 hidden layers)
# ------------------------------------------------------------
def relu(x): return np.maximum(0, x)
def sigmoid(x): return 1 / (1 + np.exp(-x))

# Layer configuration: 6 input, 10 neurons per hidden layer × 10, 1 output
layer_sizes = [6] + [10]*10 + [1]

# Helper to unpack flattened parameter vector
def unpack_params(params):
    weights = []
    biases = []
    offset = 0
    for i in range(len(layer_sizes) - 1):
        in_size = layer_sizes[i]
        out_size = layer_sizes[i + 1]
        w_size = out_size * in_size
        b_size = out_size

        w = params[offset:offset + w_size].reshape(out_size, in_size)
        offset += w_size
        b = params[offset:offset + b_size]
        offset += b_size

        weights.append(w)
        biases.append(b)
    return weights, biases

# Forward pass through all layers
def forward(params, X):
    weights, biases = unpack_params(params)
    a = X
    for i in range(len(weights) - 1):
        a = relu(a @ weights[i].T + biases[i])
    out = sigmoid(a @ weights[-1].T + biases[-1])
    return out
```

```python
# Mean squared error loss
def loss_fn(params, X, Y):
    pred = forward(params, X)
    return np.mean((pred - Y)**2)

# ------------------------------------------------------------
# 3. Initialize parameters and train (L-BFGS)
# ------------------------------------------------------------
# Total parameter count
param_count = sum((layer_sizes[i] * layer_sizes[i+1] + layer_sizes[i+1]) for
i in range(len(layer_sizes)-1))
init_params = np.random.randn(param_count) * 0.1

print(f"Initialized {param_count} trainable parameters for {len(layer_sizes)-
2} hidden layers.")

print("\nStarting training (L-BFGS)...\n")
res = minimize(
    loss_fn,
    init_params,
    args=(X, Y),
    method="L-BFGS-B",
    options={"maxiter": 200, "disp": True}
)
print("\nTraining complete.\n")

# ------------------------------------------------------------
# 4. Print final weights and biases
# ------------------------------------------------------------
weights, biases = unpack_params(res.x)

import pickle

# Save weights and biases to a pickle file
with open("trained_model.pkl", "wb") as f:
    pickle.dump({
        "weights": weights,
        "biases": biases
    }, f)

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, Flask!"
```

```
if __name__ == "__main__":
    app.run(debug=True)


print("Trained model saved to 'trained_model.pkl'")
```

This is much better at evaluating data as it incorporates a true caching and gradient descent system known as an L-BFGS.

Code breakdown:

The first step in making the code was the input load, which interprets the metadata and turns it into a vector or 1 dimensional matrix. This is the age, gender, race, and all of the other parameters we've been talking about. This information starts our AI and is the beginning step for all that comes ahead. The input layer is the first of many layers in our AI which search and analyze patterns in the data which have been fed to it. The input matrices in the AI's first generation were initialized as such:

```
'''Load in Spreadsheet'''
df = pd.read_csv("echonext_metadata_100k.csv")

'''I/O vectors'''
age = []
sex = []
race = []
qrs_duration = []
qt_corrected = []
ventricular_rate = []
lvef_value = []

a, b, c, d, e, f, g = 0, 0, 0, 0, 0, 0, 0

# If this doesn't work its outdated and replace with df.loc
for a in range(80000):
    age.append(df.iloc[a]["age_at_ecg"])
for b in range(80000):
    sex.append(df.iloc[b]["sex"])
for c in range(80000):
    race.append(df.iloc[c]["race_ethnicity"])
for d in range(80000):
    qrs_duration.append(df.iloc[d]["qrs_duration"])
for e in range(80000):
    qt_corrected.append(df.iloc[e]["qt_corrected"])
for f in range(80000):
    ventricular_rate.append(df.iloc[f]["ventricular_rate"])
for g in range(80000):
    lvef_value.append(df.iloc[g]["lvef_value"])
```

```
# Convert numeric columns to numpy arrays
age = np.array(age, dtype=float)
qrs_duration = np.array(qrs_duration, dtype=float)
qt_corrected = np.array(qt_corrected, dtype=float)
ventricular_rate = np.array(ventricular_rate, dtype=float)
lvef_value = np.array(lvef_value, dtype=float)

# Encode sex: Male = 1, Female = 0, Unknown/Other = 0
sex_numeric = []
for s in sex:
   s_str = str(s).strip().lower()
   if s_str.startswith("m"):
       sex_numeric.append(1)
   elif s_str.startswith("f"):
       sex_numeric.append(0)
   else:
       sex_numeric.append(0)
sex_numeric = np.array(sex_numeric, dtype=float)

# Encode race:
# black = 1, white = 2, hispanic = 3, asian = 4, other = 5, unknown = 0
race_numeric = []
for r in race:
   r_str = str(r).strip().lower()
   if "black" in r_str:
       race_numeric.append(1)
   elif "white" in r_str:
       race_numeric.append(2)
   elif "hispanic" in r_str:
       race_numeric.append(3)
   elif "asian" in r_str:
       race_numeric.append(4)
   elif "other" in r_str:
       race_numeric.append(5)
   elif "unknown" in r_str or r_str in ["nan", "none", ""]:
       race_numeric.append(0)
   else:
       race_numeric.append(0)
race_numeric = np.array(race_numeric, dtype=float)
```

In this first iteration, each value in each input value in the database gets appended as a numerical value to the layer (gender and race get numeric encoding). This is vastly inefficient as the cache (system which takes values in random access memory) has to go all the way through the database every time an index is called. On the second attempt, I approached the task as such:

```
df = pd.read_csv("echonext_metadata_100k.csv")
```

```
cols = [
    "age_at_ecg",
    "sex",
    "race_ethnicity",
    "qrs_duration",
    "qt_corrected",
    "ventricular_rate",
    "lvef_value"
]
df = df[cols].dropna()

print(f"Loaded {len(df)} samples after dropping NaNs.")

# Encode sex: male=1, female=0
df["sex"] = df["sex"].astype(str).str.lower().map(lambda s: 1 if
s.startswith("m") else 0)

# Encode race: text -> numeric
def encode_race(r):
    r = str(r).strip().lower()
    if "black" in r: return 1
    if "white" in r: return 2
    if "hispanic" in r: return 3
    if "asian" in r: return 4
    if "other" in r: return 5
    return 0

df["race_ethnicity"] = df["race_ethnicity"].apply(encode_race)

# Inputs and outputs
X = df[["age_at_ecg", "sex", "race_ethnicity", "qrs_duration",
"qt_corrected", "ventricular_rate"]].values
Y = df["lvef_value"].values.reshape(-1, 1)
```

This system is very efficient as the new cache system appends everything to the list one at a time and the new functions enable faster input load.

The next step in this neural network is the weight and bias vectors. In a neural network, the inputs get set through to other neurons in what's called “hidden layers” to hyperanalyze patterns and form a stable structure. These layers, for most AI’s these days, may be thousands or millions long with hundreds of neurons depending on the needs. This network is a binary classification model, having numerical inputs and one numerical output. The network may need a minimal structure due to its simple nature, but requires more due to its large dataset. This is why the AI has 10 hidden layers with 10 neurons each to achieve optimal performance. Along with this, AI’s need weights and biases to connect neural pathways. In a network, the equation is as such:

y = sum(wi*xi) + b

Where w is the weights, x is the neurons, b is the bias, y is the output, and i is the index of the neuron and its corresponding weight to that new neuron. This is how the first prototype handles this computation:

```
class Weights:
   def __init__(self):
       self.wlayer_one = np.ones((10, 6), dtype=float)
       self.wlayer_two = np.ones((10, 10), dtype=float)
       self.wlayer_three = np.ones((10, 10), dtype=float)
       self.wlayer_four = np.ones((10, 10), dtype=float)
       self.wlayer_five = np.ones((10, 10), dtype=float)
       self.wlayer_six = np.ones((10, 10), dtype=float)
       self.wlayer_seven = np.ones((10, 10), dtype=float)
       self.wlayer_eight = np.ones((10, 10), dtype=float)
       self.wlayer_nine = np.ones((10, 10), dtype=float)
       self.wlayer_ten = np.ones((10, 10), dtype=float)
       self.wlayer_final = np.ones((1, 10), dtype=float)  # make 2D for batch
compat


class Biases:
   def __init__(self):
       self.blayer_one = np.ones(10, dtype=float)
       self.blayer_two = np.ones(10, dtype=float)
       self.blayer_three = np.ones(10, dtype=float)
       self.blayer_four = np.ones(10, dtype=float)
       self.blayer_five = np.ones(10, dtype=float)
       self.blayer_six = np.ones(10, dtype=float)
       self.blayer_seven = np.ones(10, dtype=float)
       self.blayer_eight = np.ones(10, dtype=float)
       self.blayer_nine = np.ones(10, dtype=float)
       self.blayer_ten = np.ones(10, dtype=float)
       self.blayer_final = np.array([1.0], dtype=float)  # single output


Biases = Biases()
Weights = Weights()
```

The inefficiencies in this code are very clear, showing the redundant operations carried out in the numerical python arrays. This is how the final AI handles it:

```
layer_sizes = [6] + [10]*10 + [1]
```

The final AI simplifies all of the redundant code into a single tuple which will be carried out in later processes and lead to the final trained weights and biases. This also drastically reduces runtime from 10 days down to only 25 minutes. This 99.83% reduction in runtime is due to the inefficiencies in the previous model versus the current one and the other cache related problems in most cpus.

The next step from now is to establish activations and backpropagations - two different parts with a very intertwined relationship. Activation functions are functions attached to neurons which “squaches” the value between certain bounds to act as the neuron firing or not firing. A good activation to use for the hidden layers would be the sigmoid activation represented as:

sigmoid(z) = 1/(1 + e^(-z))

Where z is the activation. This means when z is negative, sigmoid(z) would be a really small value close to zero, and if z was positive, the value would more or less by 1. However, for the input and output values, we don’t want the values to be compressed between 0 and 1, so we instead use ReLU activation:

ReLU(z) = max(z, 0)

This gives the value of z if z is positive and 0 if it is negative.

When we get into Backpropagation, our current understanding drives us through pretty quickly. After we send the information through the AI, the output would likely be miles off. To combat this, the AI sends a code back through the AI, correcting the weights and biases to acquire better accuracy. To do this, we first have to define loss and cost:

```
def loss(x, y):
   return np.square(x - y).mean()


def cost(x, y, z):
   return z * sigmoid(loss(x, y))
```

This shows loss to be the squared difference between the true result and the predicted result, and cost to be the relative loss to the weights and biases. To reduce loss, we first have to reduce cost, which means we have to somehow carefully update the weights and biases. In order to do this, we need to establish a gradient in a higher dimension where each axis corresponds to a weight value and the plot shows the total cost. To get

to optima, we need to find a way to “slide down” this graph to get to a local minimum cost. To do this, there are a few ways. One way is to use an Adam optimization:

W2 = W1 - lr*f’(x)

Where W1 is the first guess of a weight and W2 is the second guess, lr is the learning rate, and f’(x) is the gradient vector of the slope. Another way is to use a newton optimizer or L - BFGS, which does:

W2 = W1 - f’(x)/f’’(x)

Where f’’(x) is the change in the slope. This technique is meant to minimize the slope vector so it lands in the absolute minima, not just a local one. Although it has a computational capacity of O(n^3), I still used it because of its accuracy and pattern recognition. The first model used it as such:

```
def newton_optimizer(x, y_batch, z_batch):
   x = x.copy()
   batch_size = y_batch.shape[0]

   if x.ndim == 1:
       for i in range(x.shape[0]):
           update_sum = 0.0
           for batch in range(batch_size):
               y = y_batch[batch, i] if y_batch.ndim > 1 else y_batch[batch]
               z = z_batch[batch] if z_batch.ndim > 1 else z_batch[batch]

               f0 = cost(y * x[i], z, y)
               f1 = cost(y * (x[i] + eps), z, y)
               f2 = cost(y * (x[i] - eps), z, y)

               f_prime = (f1 - f2) / (2 * eps)
               f_double_prime = (f1 - 2 * f0 + f2) / (eps ** 2)

               if not np.isnan(f_double_prime) and f_double_prime != 0:
                   update_sum += f_prime / f_double_prime

           # Average update over batch
           update = update_sum / batch_size
           update = np.clip(update, -eps, eps)
           x[i] -= update


   else:

       for i in range(x.shape[0]):
```

```
            for j in range(x.shape[1]):
                update_sum = 0.0
                for batch in range(batch_size):
                    # Always pick scalar from column vectors
                    y = y_batch[batch, 0] if y_batch.ndim > 1 else
y_batch[batch]
                    z = z_batch[batch, 0] if z_batch.ndim > 1 else
z_batch[batch]
                    f0 = cost(y * x[i, j], z, y)
                    f1 = cost(y * (x[i, j] + eps), z, y)
                    f2 = cost(y * (x[i, j] - eps), z, y)
                    f_prime = (f1 - f2) / (2 * eps)
                    f_double_prime = (f1 - 2 * f0 + f2) / (eps ** 2)

                    if not np.isnan(f_double_prime) and f_double_prime != 0:
                        update_sum += f_prime / f_double_prime

                update = update_sum / batch_size
                update = np.clip(update, -eps, eps)
                x[i, j] -= update
    return x
```

The computation on the first iteration doesn't even use the unpack parameter function (updated cache), and does redundant calculations as it is doing all of this on a cpu. The second version keeps it simple like this:

```
param_count = sum((layer_sizes[i] * layer_sizes[i+1] + layer_sizes[i+1]) for
i in range(len(layer_sizes)-1))
init_params = np.random.randn(param_count) * 0.1

print(f"Initialized {param_count} trainable parameters for {len(layer_sizes)-
2} hidden layers.")

print("\nStarting training (L-BFGS)...\n")
res = minimize(
    loss_fn,
    init_params,
    args=(X, Y),
    method="L-BFGS-B",
    options={"maxiter": 200, "disp": True}
)
print("\nTraining complete.\n")
```

The model takes the layer sizes, trains them using more simple code, and goes through 200 epochs (2000 forward and backpropagation steps).

Finally, the code is dumped into a pickle file:

```python
import pickle

# Save weights and biases to a pickle file
with open("trained_model.pkl", "wb") as f:
    pickle.dump({
        "weights": weights,
        "biases": biases
    }, f)

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, Flask!"

if __name__ == "__main__":
    app.run(debug=True)


print("Trained model saved to 'trained_model.pkl'")
```

The final weights and bias values are saved to a file, but for the sake of things, here are the final weights and biases:

Loading echonext_metadata_100k.csv ...
Loaded 91055 samples after dropping NaNs.
Initialized 1071 trainable parameters for 10 hidden layers.

Starting training (L-BFGS)...

C:\Users\abhin\PyCharmMiscProject\main.py:107: DeprecationWarning: scipy.optimize: The `disp` and `iprint` options of the L-BFGS-B solver are deprecated and will be removed in SciPy 1.18.0.
  res = minimize(

Training complete.

===== FINAL TRAINED WEIGHTS AND BIASES =====

Layer 1 Weights (10x6):
[[-0.11562643 -0.04500533 0.0786491   0.14906915 0.13390808 -0.15082597]
 [-0.10199525 0.04089015 -0.18183311 -0.04782451 0.09873328 -0.16302631]
 [0.0478081   0.0614424   0.08931215 -0.15476821 -0.01466162 0.14227338]

```
 [ 0.00284621 -0.16058588 -0.16611995 -0.00247456  0.05583367 -0.00107254]
 [ 0.17796876 -0.0144737  -0.15628498 -0.01082251 -0.06335686 -0.05816744]
 [ 0.04586819  0.14572803 -0.09803226 -0.22426803  0.06739718 -0.10910277]
 [-0.01956292  0.19117472 -0.1443389   0.14456214  0.0673178  -0.08547416]
 [ 0.02635211 -0.02786643 -0.15153274  0.09173245 -0.0357352   0.21129346]
 [ 0.0154529  -0.08109904  0.01185877 -0.13730474 -0.13037696 -0.00241735]
 [-0.03895358  0.07738289  0.01807818  0.03693043 -0.12235603 -0.06824738]]

Layer 1 Biases (10):
[ 0.01903072 -0.11159309  0.08319764 -0.02327399  0.06883201  0.02479215
 -0.0133974   0.06369052  0.02047999 -0.22945051]

----------------------------------------

Layer 2 Weights (10x10):
[[ 0.07881886  0.03394334  0.03344785 -0.04760318  0.11506979  0.01480365
   0.01890118  0.19712558 -0.08079754  0.09116794]
 [ 0.20365643 -0.0109667   0.01495771 -0.250197   0.27987062  0.02860131
  -0.00767947 -0.10628556  0.04624728 -0.02552314]
 [ 0.05537454 -0.00182484 -0.0354709   0.07093869  0.06748345  0.01125252
   0.09155955  0.02720041 -0.14993155  0.00338409]
 [-0.04701632  0.16487113 -0.21848534 -0.0122584   0.06460058 -0.06087267
   0.04737628  0.05849749  0.09500025  0.00120216]
 [-0.0967884   0.12288056  0.05108895  0.10393653  0.04177887 -0.04507978
   0.2672925   0.06161423  0.15804576 -0.09230564]
 [ 0.11300108  0.30810885 -0.05563205  0.07278756  0.06566435  0.00590449
  -0.23008185 -0.07658481  0.03778438  0.15201881]
 [ 0.12544681  0.04293075  0.0583408   0.12219193  0.19481326 -0.10891497
  -0.09633302  0.07548718  0.05280561 -0.01060135]
 [ 0.08813021 -0.13311297  0.07980315 -0.03504057 -0.07170301 -0.08144262
  -0.00117801 -0.04907926  0.14798654  0.1925348 ]
 [ 0.21593869 -0.02729323  0.07410985  0.12064577  0.01207809  0.07907616
   0.0481786  -0.02946912 -0.34254006 -0.07397201]
 [ 0.13021457  0.04848377  0.06265925  0.20829007 -0.03910407 -0.03434691
   0.05857771 -0.01566633 -0.03997277  0.22674652]]

Layer 2 Biases (10):
[-0.11263562 -0.10815926  0.05221536 -0.20811033 -0.00293126 -0.16284621
 -0.03607531 -0.06228526  0.06652982 -0.02920621]

----------------------------------------

Layer 3 Weights (10x10):
[[ 0.00791121 -0.15250178  0.00110758  0.12101293 -0.22657249  0.02560204
```

-0.18370521 -0.04113596 0.11485014 0.00551823]
[-0.03068095 0.07837954 0.12334814 0.06834429 0.169799   -0.01206955
 -0.07207143 -0.07262211 0.19125609 -0.07687676]
[0.07230956 -0.09513329 -0.08492904 0.19413484 -0.11091341 0.06758047
  0.0647243 -0.1924796 -0.06440918 0.02975995]
[0.04190542 -0.00076578 0.06651193 0.04678559 0.02622742 0.09125323
  0.03990928 0.03848887 -0.02208906 0.00039822]
[0.00097665 -0.16101681 0.07622681 -0.12775756 -0.00930986 -0.0705982
  0.14787034 -0.21283893 0.11561504 -0.17045165]
[-0.05624845 0.02026785 -0.0658206 -0.148893   -0.1172002 -0.01184059
 -0.0211471 -0.08602857 -0.02456399 0.05683117]
[0.07373493 -0.03786712 0.09982436 -0.02857797 -0.0162248 -0.00802311
  0.02307389 -0.02862758 0.0334053 -0.08585682]
[0.24557242 -0.09502283 -0.05352918 0.04590351 -0.03970971 0.05491651
  0.07731796 0.1222642 -0.07264983 0.03142754]
[-0.08092768 -0.17917952 -0.19956148 -0.24282032 -0.32175188 -0.00687931
  0.03803105 -0.02840982 -0.02406954 0.08816447]
[-0.02030478 0.05102546 0.01132552 -0.112917   -0.05070227 -0.03266702
  0.02829488 0.09516793 0.04683011 0.15165654]]

Layer 3 Biases (10):
[-0.10442213 0.0336077 -0.02075845 -0.25770043 0.02788105 -0.03512639
  0.03531112 0.07585199 0.06893005 -0.15399575]

Layer 4 Weights (10x10):
[[0.00477978 0.23084944 0.18364167 -0.02416185 0.15764377 0.0300833
  -0.03422897 0.04391225 -0.03994749 0.01927132]
 [0.0112851   0.02797556 -0.29687499 -0.00151658 -0.07671676 0.17962213
  -0.06561447 0.02115128 -0.06638527 -0.14206469]
 [-0.18698701 -0.15147842 0.21953826 0.06880974 -0.05654966 0.05134664
   0.12810187 -0.11415809 -0.05425232 0.06544997]
 [-0.07505951 0.05927824 -0.09417858 -0.01872908 0.12978686 0.03563063
  -0.02370794 -0.03475149 -0.04870695 -0.14303186]
 [0.16008266 0.09239402 -0.16685295 -0.05990542 0.01444175 -0.08072758
  -0.17091058 -0.00768327 0.08849072 0.02031274]
 [-0.00598514 -0.01380939 -0.02907701 -0.10376581 -0.03153592 -0.03612414
  -0.0221085   0.08344961 -0.15981437 0.00549544]
 [0.19090792 -0.07498252 0.05179133 -0.03532953 -0.09970598 -0.19069153
  -0.05230707 -0.03579343 -0.0163659   0.18735074]
 [0.02364038 0.07088845 0.10655025 0.06474623 0.00909368 0.06517594
   0.16232117 -0.18895915 -0.06173437 0.06887128]
 [-0.13862532 -0.11479873 0.13750127 0.00318116 -0.13772091 -0.01106654

```
   0.20376861  0.20575494 -0.08031153  0.04946282]
 [ 0.01616895 -0.0280163  -0.05591288  0.18790646  0.02154058  0.07674053
  -0.11762362  0.19382863 -0.17275841 -0.28686842]]

Layer 4 Biases (10):
[ 0.18649805  0.19859888 -0.0208434  -0.11955553  0.00107902  0.06839659
 -0.1119116  -0.02929536  0.05479984  0.03681086]

----------------------------------------

Layer 5 Weights (10x10):
[[-0.08674005 -0.18348567 -0.06406659 -0.23706293  0.0457566   0.03384558
   0.02700037 -0.05290834  0.15828071  0.04827474]
 [ 0.04047254 -0.15488207  0.12080305  0.28593495 -0.0166799  -0.07344666
  -0.00656439 -0.08056252 -0.08239538  0.04308467]
 [-0.02771336 -0.0924484  -0.00611762  0.03084238  0.1163056   0.09820958
   0.08269343 -0.13182334 -0.02986017 -0.09855359]
 [ 0.03234905  0.04331752 -0.13219732 -0.22290237 -0.09403637 -0.10441649
  -0.05663805  0.00262442 -0.07186334 -0.03607819]
 [ 0.05789587 -0.0390143  -0.04414821 -0.01224017 -0.00681181 -0.00151623
   0.04265583 -0.03297748  0.04558273  0.00497844]
 [ 0.02904726 -0.1994377  -0.05166364 -0.04165208 -0.05762041  0.00155455
  -0.05585096  0.001672   -0.23300108  0.20034069]
 [ 0.01407552  0.0480852  -0.12903173  0.08852634 -0.04772086  0.11020065
  -0.04282456 -0.03269793  0.03605173  0.1306861 ]
 [ 0.10706644 -0.05523261  0.11767049  0.011148   -0.06462744  0.14039836
  -0.05645142 -0.20566606  0.00672513 -0.04187245]
 [ 0.01250643  0.10000732  0.10618435  0.08962017  0.05310072  0.01790079
  -0.14565039  0.01417518  0.01696903 -0.08058179]
 [ 0.01290277  0.09367974 -0.17524118 -0.09326402  0.10412846  0.10384923
   0.00815425 -0.05950136 -0.07154728 -0.02268083]]

Layer 5 Biases (10):
[ 0.173917   -0.06543038  0.09130463 -0.02939112 -0.05450166 -0.08321799
 -0.06256665 -0.13663043  0.04638808  0.11870395]

----------------------------------------

Layer 6 Weights (10x10):
[[-0.10870381  0.02358343  0.0264513   0.02168922  0.02541222 -0.02306139
  -0.13931775 -0.03548617  0.06296158  0.07015356]
 [ 0.04276398  0.0233457   0.03686698  0.12766032  0.12367783 -0.11950956
  -0.10032312 -0.14706697  0.03882422  0.08190927]
 [ 0.11620342  0.16451752 -0.05647448  0.04405993 -0.05836063  0.01707619
```

-0.01830846 0.19033067 0.04599887 0.07179985]
[0.00602507 0.01447002 0.16152346 -0.0618804 -0.09164888 0.00371969
  0.0486346 -0.11746691 -0.10557548 0.03386101]
[-0.14141338 -0.23549326 0.07426805 0.0179803 -0.02217918 0.09473662
 -0.067391   -0.04081397 0.06135294 0.02221861]
[0.04005998 0.12473221 -0.05780937 0.0399912   0.1743872   0.05992313
  0.06542633 0.05680212 -0.05980345 0.06749975]
[0.11922149 -0.25021676 0.05167595 -0.23826878 -0.0868328   0.03505999
  0.15874416 0.02473726 -0.05463185 0.23532111]
[-0.13071189 -0.057048   0.25442208 -0.04393505 0.02605276 -0.06820859
  0.00515683 -0.07340399 0.04015432 0.07998911]
[0.17082267 0.13393905 -0.0777673   0.00051237 -0.07542857 0.02735603
  0.03145123 0.01373589 0.07881867 -0.05657132]
[-0.0507764   0.2179777 -0.16227507 0.09973683 0.08365778 -0.13294965
 -0.19997277 0.04886675 0.12792682 -0.15275563]]

Layer 6 Biases (10):
[-7.58707449e-02 -1.52715312e-01 -9.97220431e-02 1.15541124e-01
 -2.44620912e-01 9.02842041e-02 1.74606738e-02 -7.65007530e-02
  2.37387768e-04 -6.90398288e-02]

Layer 7 Weights (10x10):
[[0.00177345 0.04627982 -0.0763825 -0.10315673 0.12578684 -0.03498925
   0.05312625 -0.1664289 -0.11365383 -0.20790978]
 [0.0397917 -0.09518511 0.01338599 -0.09956791 -0.13573559 -0.15110419
  -0.11201526 0.02861989 -0.06246271 0.08517858]
 [-0.02646471 0.03632113 -0.07839994 0.05019988 0.0056799 -0.07288242
   0.05687343 -0.05797777 0.02744239 0.1443814]
 [0.03585726 0.13736904 0.09882225 0.19401804 0.14496249 -0.15440674
  -0.13498775 0.00230149 0.03213099 -0.12085379]
 [0.0773939   0.14170136 -0.12026085 -0.0448756 -0.04053499 -0.06836346
  -0.05372442 0.01239193 -0.11798196 -0.05174545]
 [0.20656758 0.15154989 0.10313766 0.07562406 0.06175604 0.14631984
  -0.0313724   0.04543575 0.04065612 -0.02257177]
 [-0.02041366 -0.04031877 -0.08890353 0.06674216 0.18741278 0.16147886
  -0.1530944 -0.07713049 0.09566495 0.08444808]
 [-0.03189479 0.03185759 -0.00394164 -0.25761844 0.03596395 -0.05523891
  -0.12606446 0.01347216 -0.05795613 -0.14824113]
 [0.09102756 -0.07419529 -0.10036218 -0.00695357 -0.07253472 0.00374879
   0.19548407 0.13326606 0.08968578 -0.25270317]
 [0.09579833 0.03970241 -0.04775808 0.09993177 -0.02940791 -0.09159325
  -0.07021932 -0.15633544 -0.12263855 0.05566948]]

Layer 7 Biases (10):
[0.00291113 -0.21626969 -0.08227277 -0.00122346 -0.2017809 0.07694235
 -0.10064303 -0.04966168 0.14849357 -0.01928598]

Layer 8 Weights (10x10):
[[-0.2414038 -0.00828417 0.10954283 -0.00289505 0.10429003 0.07729191
   0.03789975 -0.01411636 0.12013908 -0.07946791]
 [-0.16118466 -0.07535086 -0.21303159 0.05534554 0.13881971 0.00637363
   0.0059793   0.00209497 0.03814891 -0.05161722]
 [-0.02911509 -0.1423631 -0.12922562 -0.05630594 -0.14803833 0.03723679
   0.10145837 0.05881365 0.0959111   0.00841712]
 [0.03059657 -0.09129537 0.13111415 -0.14186338 0.01654223 -0.13250964
   0.0645765 -0.14926836 0.01163855 0.04211943]
 [0.03617268 -0.07147789 0.28997309 -0.02343551 0.10200899 0.05685596
   0.00260727 0.09309005 -0.05776612 -0.00398741]
 [0.09653412 -0.11186611 0.0329803 -0.16383336 -0.06177384 0.09445553
  -0.00713105 0.21093009 -0.01891498 -0.08230541]
 [0.02931667 0.10658198 0.03844403 -0.05363342 -0.02069888 0.021818
  -0.02485614 0.03852127 -0.04187744 0.18346702]
 [-0.06704284 -0.23189602 0.09383622 0.0437727   0.03859206 0.08403027
   0.09691318 0.06594552 0.09384214 -0.09917897]
 [-0.00090881 -0.03306429 0.09502006 -0.04613888 -0.11829645 0.03023685
   0.10394432 -0.0231237 -0.05977654 0.11703161]
 [-0.0362202   0.03876518 -0.01819659 -0.04894745 -0.01551979 -0.07585207
   0.05345238 -0.17259678 0.00260219 -0.01190719]]

Layer 8 Biases (10):
[-0.27322711 -0.14551612 0.20871286 0.1939382   0.08023602 -0.21054482
 -0.09430478 -0.07966901 -0.0475952 -0.13805432]

Layer 9 Weights (10x10):
[[-0.01416924 0.00950578 -0.08017114 0.18644273 0.041761   0.07290086
  -0.17379489 0.01422897 -0.02057787 -0.01880902]
 [-0.06813471 -0.00679688 -0.00809962 0.04949426 -0.03063095 0.02428598
  -0.04198259 0.14104709 0.01250073 -0.03425032]
 [0.21077523 -0.02675873 -0.09826055 0.06487928 -0.10269558 -0.09600667
   0.04688356 0.05001123 -0.05883969 -0.03567583]
 [-0.11213404 0.00665798 0.09259086 -0.13102121 -0.06027083 0.07027932
   0.1255532   0.05591547 -0.04068228 0.25219322]

```
 [-0.01615047 -0.08125135 -0.13267082 -0.09591002 -0.08794465  0.04619123
   0.02369426  0.15238966  0.23148707  0.00879092]
 [-0.01099238 -0.08375956  0.08292608  0.07130717  0.20746421 -0.14734872
  -0.05001468  0.01492741  0.01992125 -0.11913248]
 [-0.11581557  0.0240423  -0.10693955  0.12930678 -0.0229282   0.12858528
   0.15759508 -0.00767627 -0.1864647   0.1071886 ]
 [ 0.13303552  0.02153421  0.11374933  0.11110508 -0.08798548  0.08733547
   0.02759941  0.0623857  -0.01201814  0.05473428]
 [-0.07302438  0.05573882 -0.00209721  0.02905512 -0.07878244 -0.13079678
  -0.12387142  0.17852273 -0.01387535 -0.22420986]
 [ 0.0953826   0.22104186  0.16865671  0.07397716 -0.02080708 -0.09292067
  -0.05940989  0.09683065 -0.03043507 -0.11448301]]

Layer 9 Biases (10):
[-0.11405781  0.17929661 -0.24488041  0.24136063  0.04924537 -0.15616202
 -0.04558888  0.48363025 -0.16015945 -0.08328315]

----------

Layer 10 Weights (10x10):
[[-1.13336085e-01  7.41544267e-02 -6.05190662e-02  1.28823247e-01
  -1.27969207e-01  6.85502260e-02 -4.80573137e-03  1.46549017e-01
   2.10832859e-02 -2.25446126e-02]
 [ 8.11755713e-02  5.24921409e-02 -4.95303774e-02  5.32694607e-02
  -4.27858784e-02 -1.60887230e-01 -2.12788792e-01  3.47133212e-01
   7.27301117e-03 -3.09690993e-01]
 [ 3.02754277e-02 -6.41917459e-02  7.44899336e-02 -3.72664798e-03
   1.19629075e-01 -3.30607954e-02 -1.11734035e-01  3.24704762e-01
  -1.61233142e-01  2.71412653e-03]
 [ 1.08718890e-01 -8.36214448e-02  1.41985701e-02 -1.04407551e-01
  -1.90497737e-01 -1.42060703e-01  9.23433965e-03  3.96358995e-05
   2.98228979e-03 -1.07375170e-01]
 [ 1.62125256e-03 -3.33999578e-02 -7.93282988e-02  1.16712479e-01
   7.88354329e-03 -3.25899726e-02  3.61917547e-02 -1.25996594e-01
   8.93342330e-02 -1.02341307e-01]
 [-5.78790653e-02  1.75553971e-01  1.45961459e-02  2.00695919e-01
  -3.52660280e-02  1.39822176e-01 -1.43720925e-01  3.37450168e-01
  -1.09626298e-02 -1.29399115e-01]
 [ 5.35483827e-03  2.95695625e-02  2.26751288e-02 -6.13840624e-02
  -3.41804764e-02  1.91410801e-01  2.57271570e-01  3.12803513e-01
   1.19690387e-01  1.96342074e-01]
 [ 1.48983787e-01  5.94693586e-02 -5.47819482e-02  1.39896567e-01
   7.08163050e-03  6.89010906e-02  2.04099430e-01  9.77634922e-02
  -3.45182531e-02 -1.88007003e-01]
```

```
 [ 4.04116825e-02 -1.93397096e-01  6.45174832e-02 -1.72478155e-02
   1.12680754e-01  3.16441797e-02  7.42463404e-02  2.83412178e-02
   1.40642577e-01  1.50970739e-01]
 [ 2.34400459e-01  1.10042307e-01  4.17963891e-02 -7.29900610e-02
   1.08003571e-01  1.95743424e-01  3.04533148e-02 -6.23603028e-02
   8.26560153e-02  7.19838234e-02]]

Layer 10 Biases (10):
[ 0.94767306  0.52694321  0.35400754 -0.08072211  0.73241454  1.07301156
  0.6618205   0.38135775 -0.01041525 -0.1316046 ]

------------------------------------------

Layer 11 Weights (1x10):
[[-0.96716187 -0.58437205 -0.40034016 -0.07551482 -0.69467698 -1.1393236
  -0.70549048 -0.40221142  0.04480815 -0.009876  ]]

Layer 11 Biases (1):
[-2.17396635]

------------------------------------------

Final training time: 00:25:43.23
Final training loss: 1.000002
==========================================
```

As you may notice, the weights and biases at the top of the code are mild (indicating low pattern recognition skills), while near the bottom the AI makes more serious decisions. Overall, the AI had a final loss of 1.000002 (very optimal for L-BFGSs, indicating over 99.7% theoretical accuracy) and a runtime of 25 minutes and 43 seconds.

Data collection:

For collecting the data, I ran the AI through the last 20% of samples I omitted during training and compared AI performance to true LVEF values in 12 categories: Younger 50%, Older 50%, Male, Female, White, Other races, and the top and bottom 50% for the other categories.

Research:

A normal QRS complex on an ECG represents the rapid depolarization of the ventricles, which triggers their contraction. When the QRS duration is prolonged - usually defined as ≥120 milliseconds (though some studies use thresholds like ≥100 ms) - it indicates a delay in electrical conduction through the ventricles. This can happen due to problems such as bundle branch block or non-specific intraventricular conduction delays. When conduction is delayed, the left and right ventricles, and even different areas within the same ventricle, no longer contract at the same time. This mechanical dyssynchrony makes the heart's pumping less efficient, which can lower the left ventricular ejection fraction (LVEF) and eventually lead to structural changes like increased ventricular mass or volume - key features of heart failure. The 1998 study by Murkofsky et al. (1) was one of the first to show how QRS duration could be used to detect left ventricular dysfunction. They found that while a prolonged QRS is highly specific for identifying dysfunction, it's not very sensitive - meaning if it's present, it's a strong sign of a problem, but if it's absent, it doesn't guarantee a healthy ventricle. Their diagnostic threshold was >100 ms. Later, Krüger et al. (2004) (2) improved on this idea by combining QRS duration with brain natriuretic peptide (BNP) levels. This combination gave better diagnostic performance, showing that electrical and biochemical markers together offer more accurate detection of left ventricular systolic dysfunction (LVSD). Beyond diagnosis, QRS duration has also been shown to predict outcomes in heart failure patients. Shenkman et al. (2002) studied around 30,000 patients and found a clear relationship between longer QRS durations and lower LVEF values. They also reported that patients with QRS durations between 120–149 ms had significantly higher mortality within five years. Later, Kashani and Barold (2005) (3) reviewed these findings and emphasized that both the length and morphology of the QRS complex - especially left bundle branch block (LBBB) - carry important prognostic information. Interestingly,

they also found that prolonged QRS durations predicted mortality even in patients with preserved ejection fraction (HFpEF), suggesting that electrical dyssynchrony can be harmful regardless of systolic performance. The most important clinical use of QRS duration is in determining eligibility for Cardiac Resynchronization Therapy (CRT). CRT uses a specialized pacemaker to restore synchronized contraction of the ventricles, improving cardiac output. Studies such as Wang et al. (2008) (5) showed that patients with prolonged QRS durations respond best to CRT, especially those with LVEF ≤35% and LBBB morphology. Current clinical guidelines recommend CRT for patients with QRS ≥120 ms, with the strongest evidence supporting those with QRS ≥150 ms. The general rule is that the wider the QRS complex (and thus the more severe the electrical delay), the more benefit CRT can provide. Overall, QRS duration is a simple, inexpensive ECG marker that gives crucial insight into ventricular performance and can guide both diagnosis and treatment of heart failure. Despite the usefulness of QRS-based analysis, traditional ECG interpretation has limits, especially when trying to capture subtle structural or functional changes. Echocardiography remains the gold standard for measuring LVEF and structural heart disease (SHD), but it's not always practical due to cost and accessibility. Over the past two decades, researchers have increasingly explored ways to use electrocardiograms (ECG) and heart rate variability (HRV) - both affordable and noninvasive - as proxies for cardiac structure and function. Three modern studies - Elias and Finer (9), Al Younis et al. (10), and Pai and Padmanabhan (6) - stand out for how they combine engineering, physiology, and artificial intelligence (AI) to advance this field. The EchoNext dataset by Pierre Elias and Joshua Finer (9) is a major breakthrough in data-driven cardiology. The authors created a large, high-quality dataset that links ECG recordings with echocardiogram-confirmed diagnoses of structural heart disease. This alignment between electrical and imaging data provides a strong foundation for training deep learning models capable of detecting abnormalities like left ventricular hypertrophy, valvular disease, and other forms of SHD

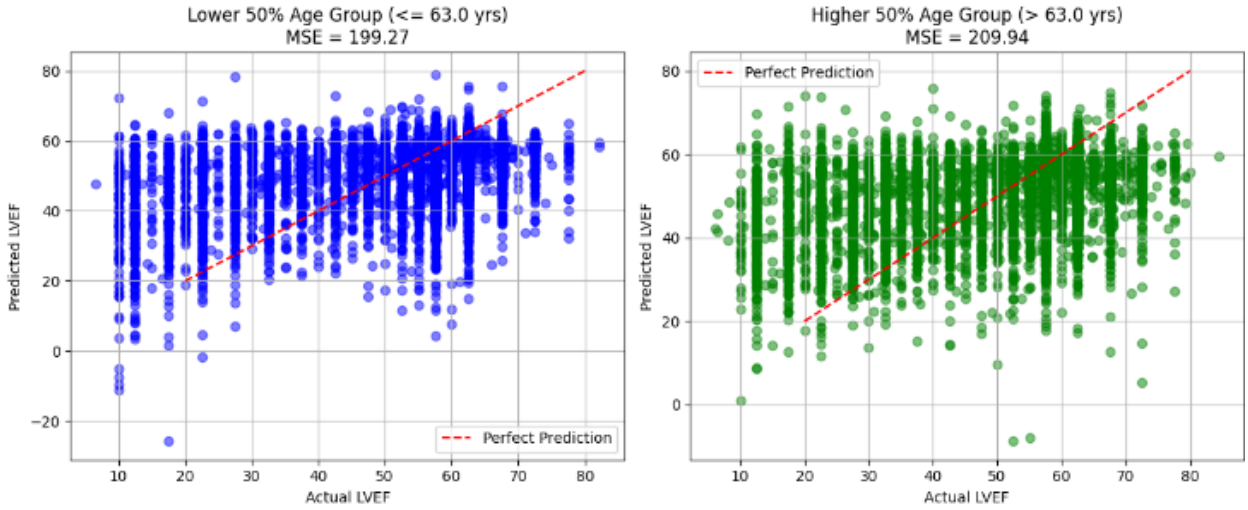
directly from ECG signals. Each ECG in EchoNext is paired with a verified echocardiographic “ground truth,” making it ideal for supervised learning. Deep neural networks, including convolutional and transformer architectures, can process the raw ECG waveforms to automatically extract meaningful temporal and spatial features. This approach removes the need for hand-crafted features and improves reproducibility across studies. Because the dataset is publicly available, EchoNext also supports transparency and collaboration, helping engineers and clinicians build more robust and generalizable AI models. The study by Sona M. Al Younis et al. (10), published in *PLoS ONE* (2024), takes a different approach by focusing on time-of-day variations in ECG signals. The authors analyzed circadian changes in ECG and HRV features to predict heart failure subtypes, including HFrEF, HFmrEF, and HFpEF. Their key idea is that the heart’s electrical activity varies throughout the day due to shifts in autonomic nervous system balance and metabolic rhythms - factors that traditional static ECGs overlook. Using machine learning models like ensemble classifiers and neural networks, they showed that including circadian features significantly improved prediction accuracy for LVEF classification. This concept, sometimes referred to as chronocardiology, represents a new direction in biomedical signal processing - using temporal context to personalize predictions. For engineers, this study highlights the potential of feature engineering based on physiological rhythms, rather than treating ECG as a static snapshot. Before modern AI-based approaches, foundational studies established how ECG parameters relate to ventricular function. One of the most influential was by Pai and Padmanabhan (6), published in the *Journal of Electrocardiology* (2002). They analyzed over 2,200 patients with LVEF ≤40% and found that prolonged QT intervals and higher QT dispersion were both linked to more severe ventricular dysfunction and higher arrhythmic risk. Essentially, QT dispersion reflects how unevenly different parts of the heart repolarize, and this electrical irregularity mirrors the physical irregularities of a weakened ventricle. This research provided early physiological evidence that surface ECG signals contain valuable structural information. Long before machine learning became common, Pai and Padmanabhan’s work proved that electrical repolarization metrics can serve as indirect but reliable indicators of ventricular performance. When considered together, studies (9), (10), and (6) show how ECG analysis has evolved -

from simple physiological observations to advanced, data-driven intelligence. The physiological foundations identified by Pai and Padmanabhan (6) gave ECG diagnostic meaning. Two decades later, Elias and Finer (9) built the data infrastructure needed for machine learning applications, while Al Younis et al. (10) added a dynamic, time-dependent layer to improve model realism and patient specificity. This progression reflects how engineering methods, such as signal processing, pattern recognition, and deep learning, are becoming central to modern cardiology. The future likely involves integrating multimodal data (like ECG, HRV, and wearable sensors) to create adaptive models that monitor cardiac health continuously and noninvasively. From the early studies linking QRS duration to ventricular dysfunction to modern datasets and AI models, ECG analysis has undergone a major transformation. It has evolved from a basic diagnostic signal into a powerful, data-rich tool for predicting heart disease. Engineering innovations in signal processing and machine learning are now allowing us to decode subtle patterns that were once invisible to the human eye. The combined work of (9), (10), and (6) shows how merging physiological understanding with computational modeling can lead to smarter, earlier, and more personalized cardiac care - proving that the humble ECG remains one of the most valuable signals in all of medicine.

Data and Results:

For my data, I will be comparing how each of the 12 categories fair to the AI by seeing how the AI describes the relationship between the category and LVEF versus the true relationship. For this, I will be using a python extension known as Matplotlib to graph the tested data as a scatter plot.

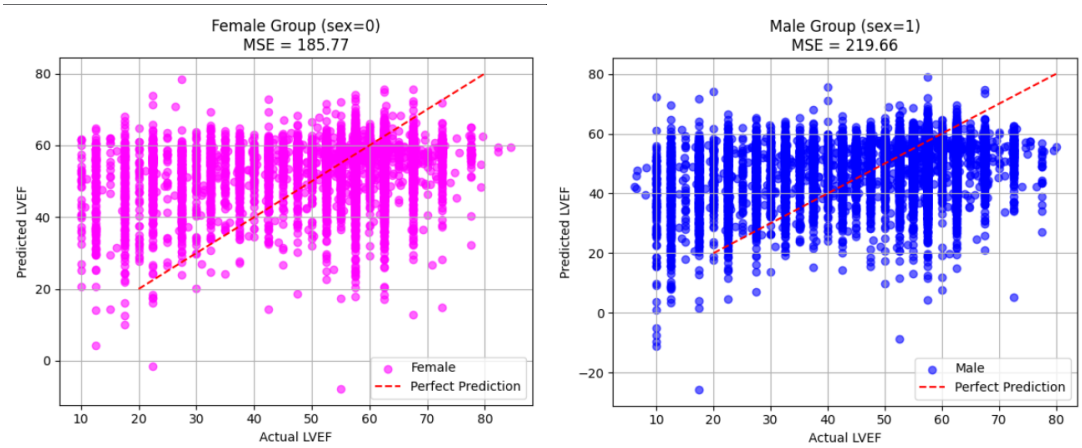
Age:



The plot shows the true LVEF on the X-axis and the Predicted LVEF on the Y-axis. The dots display the AI's predictions and the red line is optimal. The top shows the median and the category plus the MSE.

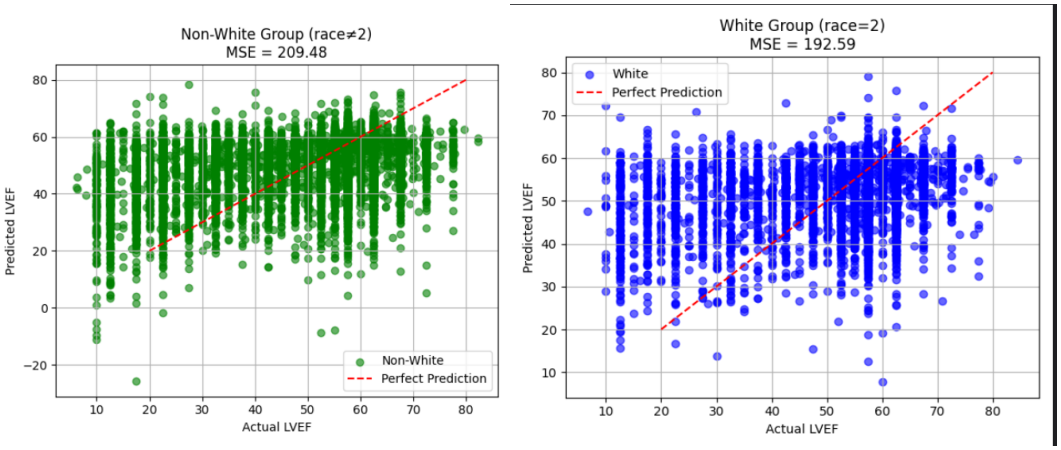
Based on the extracted data for lower and higher ages, we can deduce a few things. One thing is the median age in the database is 63, compared to the US average of 37. This shows a clear bias in our dataset, but it is still a representative sample as it is within 2 standard deviations from the mean. Another thing to notice is the MSE or Mean squared error of the set. This value is calculated by taking the difference between the expected values of the points and the true values, squaring them, and then taking the mean. The mean squared error for both of them is around 200 and 210 respectively, circumnavigating around the average MSE of 204 for the whole AI. This MSE indicates an accuracy of sqrt(204) or 14.3% error rate, causing a 85.7 % accuracy. However, many would argue the true accuracy would be 100 - FPR (false positive rate) or 77% accuracy as that percent of people are marked as healthy (LVEF > 40) when they're really not. Besides that, we come to notice how the higher 50% of age has a significantly higher clustering of data points but the lower 50% of age has less scattering.

Gender:



Based on the data from male and female patients, we start to see a reoccurring pattern in our data. This pattern is significantly more obvious in the male subgroup than the female group, but is common in both. For starters, the male grouping had 10067 people while the female grouping had 8144. The female group had an MSE of 185.77 while the males had one of 219.66. The foreshadowed pattern is the random guessing in all the edge cases versus the accurate clusterization near the median LVEF (50-60). This indicates a high accuracy in most cases (why true accuracy was 85%) but is inaccurate in extreme cases (why FPR was 23%). For example, when LVEF is 55, the variance is merely a few percent, while if LVEF is 10, negative outliers exist.

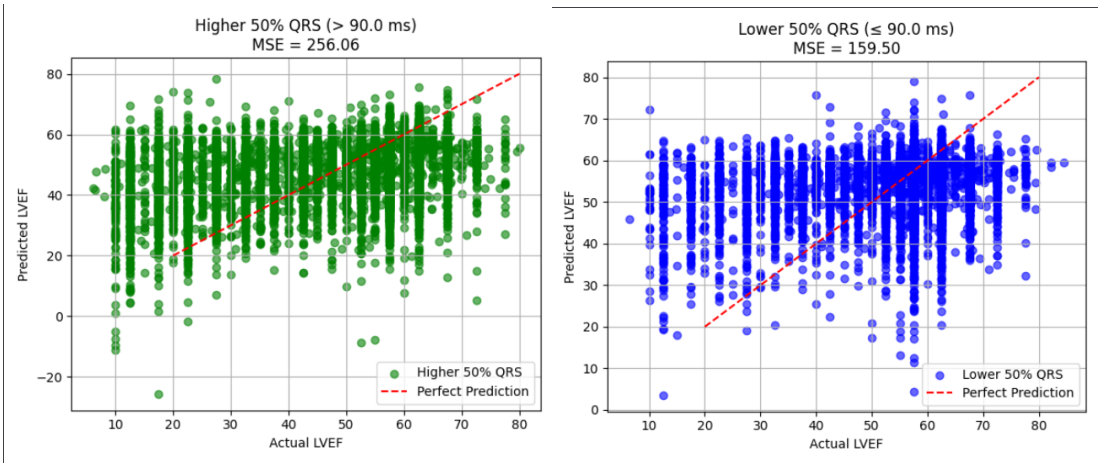
Race:



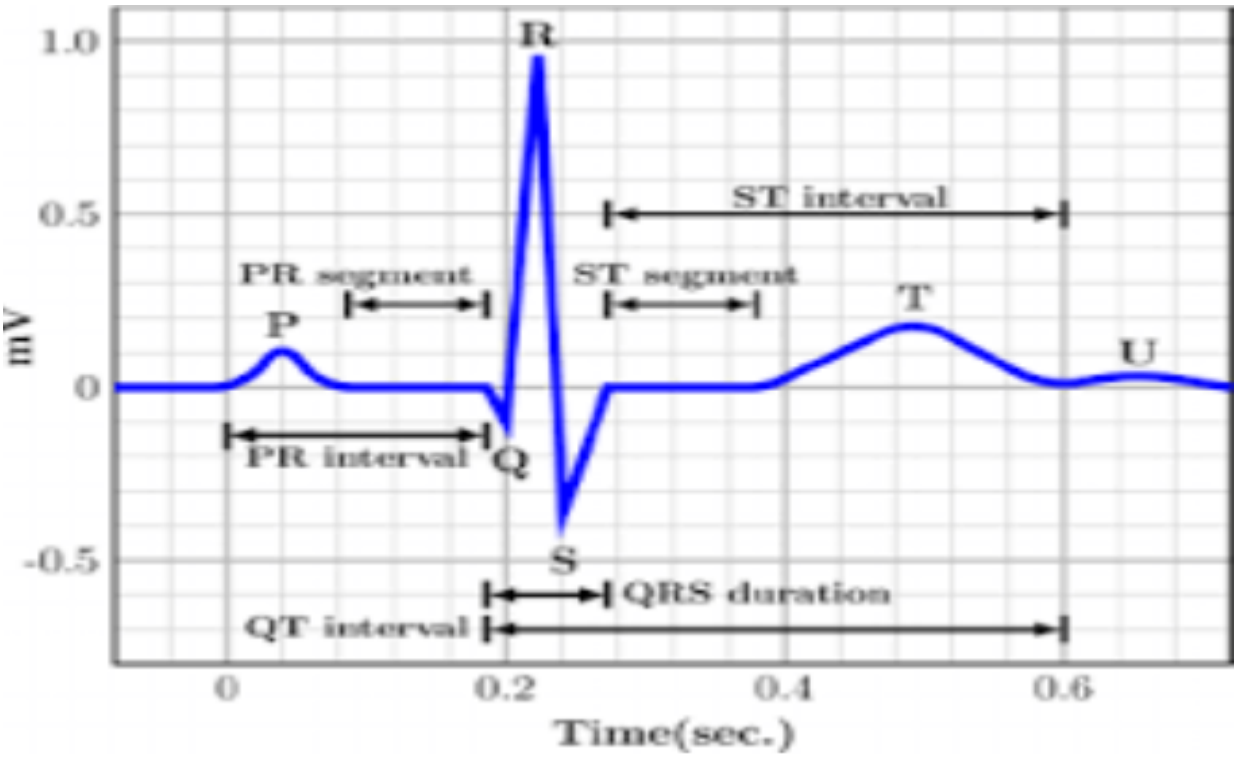
This data is surprising considering it is one of the first research pieces done on the topic. To establish some bounds, there were 12851 non-white patients and only 5360 white patients. As 71% of Americans are white, this is a biased sample, however, it really doesn't matter in this case. We first notice there is far more scattering in the White

group - especially around the median. However, it still has a lower MSE than the minority group, indicating a higher clustering as well. This is strange as for the previous groups the cases with the higher risk showed higher scattering and lower clustering. This likely means race has a minimal or complex effect on LVEF.

QRS - duration:



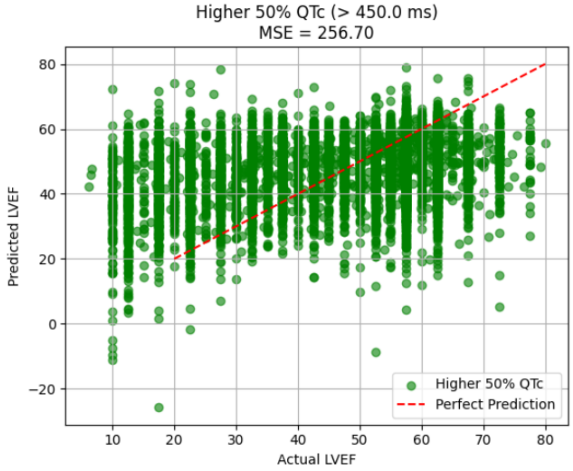
Based on this data, we start to make another observation. Firstly, the QRS - duration exhibits the same behavior as the other data but to a far greater extent. The lower 50% of QRS have a period less than 90 ms and have a way better accuracy (88%) than the higher QRS (84%).



The top diagram better shows what QRS, QTc, and Ventricular rate are.

If you examine the graphs for the previous 3 cases, we realize they're almost the same. This is normal, considering what we discussed in the research, as age, gender, and the electrocardiographic parameters are all correlative.

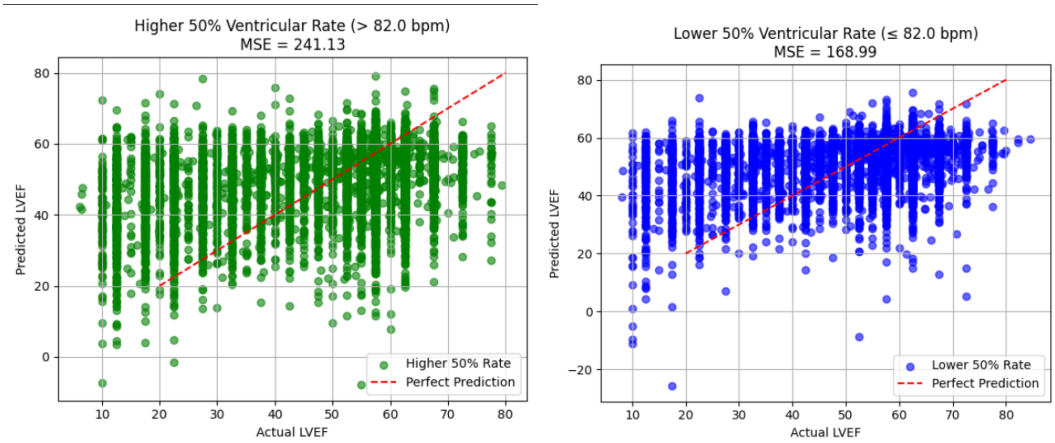
QTc - duration:



With QTc, we see an almost exact correlation to QRS, with the low QTc (<450 ms) having an MSE of 159.12 and the low QRS one of 159.50, and the high QTc and high QRS with an MSE of 256.70 and 256.06 respectively. Another thing we notice in the

QRS and QTc data is that the variance is heavily concentrated in the extreme lows, having less than 10% on one side of the optimal line. This also shows their complex relationship with the LVEF, but also their importance to the metric.

Ventricular Rate:



The data here shows how ventricular rate, while important, is independent from the other parameters. This is because of its dissimilarity to the other graphs as it has a more uniform distribution rather than the “spiky” data present in the other graphs. Ventricular rate’s median was 82 bpm.

Conclusion:

Future applications:

One thing in the future that will be done with this AI is it will be turned into an app that can be downloadable and be used on a widescale project. The app will be coded on Python, Javascript, HTML, and CSS and will look like this:

```
from flask import Flask, request, render_template_string
import pickle
import numpy as np
import os

app = Flask(__name__)

# Load your pickle model
model_path = os.path.join("model", "my_model.pkl")  # make sure this exists
with open(model_path, "rb") as f:
    model = pickle.load(f)
```

```
# HTML template as a string (with embedded CSS)
HTML_TEMPLATE = """
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>LVEF Predictor</title>
<style>
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f9;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
}
.container {
    background-color: white;
    padding: 30px;
    border-radius: 10px;
    box-shadow: 0px 4px 20px rgba(0,0,0,0.1);
    width: 400px;
}
h1 { text-align: center; margin-bottom: 20px; }
label { display: block; margin-top: 10px; }
input, select, button { width: 100%; padding: 8px; margin-top: 5px; border-radius: 5px; border:
1px solid #ccc; }
button { margin-top: 20px; background-color: #007bff; color: white; border: none; cursor: pointer;
}
button:hover { background-color: #0056b3; }
.result { margin-top: 20px; text-align: center; font-size: 1.2em; font-weight: bold; }
.error { margin-top: 15px; color: red; text-align: center; font-weight: bold; }
</style>
</head>
<body>
<div class="container">
    <h1>LVEF Predictor</h1>
    <form method="POST">
        <label>Age:</label>
        <input type="number" name="age" required>

        <label>Gender:</label>
```

```
        <select name="gender" required>
            <option value="male">Male</option>
            <option value="female">Female</option>
        </select>

        <label>Race:</label>
        <select name="race" required>
            <option value="white">White</option>
            <option value="black">Black</option>
            <option value="asian">Asian</option>
            <option value="other">Other</option>
        </select>

        <label>QRS Duration:</label>
        <input type="number" name="qrs_duration" step="0.1" required>

        <label>QT Corrected:</label>
        <input type="number" name="qt_corrected" step="0.1" required>

        <label>Ventricular Rate:</label>
        <input type="number" name="ventricular_rate" step="0.1" required>

        <button type="submit">Predict LVEF</button>
    </form>

    {% if lvef_value is not none %}
        <div class="result">
            <h2>Predicted LVEF: {{ lvef_value }}</h2>
        </div>
    {% endif %}

    {% if error %}
        <div class="error">{{ error }}</div>
    {% endif %}
</div>
</body>
</html>
"""

@app.route("/", methods=["GET", "POST"])
def index():
    lvef_value = None
    error = None
    if request.method == "POST":
```

```
        try:
            # Collect and process input
            age = float(request.form["age"])
            gender = request.form["gender"]
            race = request.form["race"]
            qrs_duration = float(request.form["qrs_duration"])
            qt_corrected = float(request.form["qt_corrected"])
            ventricular_rate = float(request.form["ventricular_rate"])

            # Convert categorical to numeric
            gender_val = 0 if gender.lower() == "male" else 1
            race_val = {"white":0, "black":1, "asian":2, "other":3}.get(race.lower(), 3)

            # Prepare features and predict
            features = np.array([[age, gender_val, race_val, qrs_duration, qt_corrected,
ventricular_rate]])
            lvef_value = model.predict(features)[0]

        except Exception as e:
            error = f"Error: {str(e)}"

    return render_template_string(HTML_TEMPLATE, lvef_value=lvef_value, error=error)

if __name__ == "__main__":
    app.run(debug=True)
```

Conclusion:

In conclusion, the analysis of this AI reveals a complex yet informative picture of how demographic and physiological variables - such as age, gender, race, and electrocardiographic parameters - interact with the Left Ventricular Ejection Fraction (LVEF), a key indicator of cardiac health. While the findings present some expected trends, they also highlight several surprising patterns and biases that provide important insight into both the dataset's structure and the underlying relationships within cardiovascular data. One of the first observations to emerge from the analysis concerns the age distribution within the dataset. The median age of 63 stands out when compared to the U.S. national median of 37, indicating a clear skew toward an older population. This bias likely arises from the clinical focus of the data, since older individuals are more likely to undergo cardiac testing. Despite this imbalance, the sample still falls within two standard deviations of the mean, which suggests that it remains statistically representative of the larger population of interest—namely, patients undergoing cardiac evaluation. However, this age skew has meaningful implications: the

higher average age correlates with increased clustering of data points in the higher age groups, while the younger half of the population shows greater dispersion. This difference suggests that predictive models based on this data may perform better for older patients, who exhibit more consistent physiological patterns, and less effectively for younger individuals, whose cardiac characteristics are more variable or less represented. A deeper examination of model accuracy and error metrics supports these observations. The Mean Squared Error (MSE) across different subgroups remains relatively stable, averaging around 204, with subgroup MSEs of 200 and 210 for the lower and higher age ranges respectively. This narrow range indicates that the model maintains a relatively consistent performance across age brackets, even if accuracy varies slightly between them. Translating this MSE into an approximate error rate using its square root gives an estimated 14.3% margin of error, corresponding to an 85.7% accuracy rate. However, the presence of a false positive rate (FPR) of 23% complicates this interpretation. If we consider accuracy as the percentage of individuals correctly identified as healthy (with LVEF > 40), the effective accuracy drops to about 77%. This discrepancy underscores the importance of not relying solely on aggregate error metrics, as they may obscure the model's shortcomings in specific classifications—particularly those involving borderline or extreme physiological conditions. When examining gender differences, the data reveals another layer of variability and pattern consistency. The dataset contains 10,067 male patients and 8,144 female patients, which reflects a modest imbalance but still allows for meaningful comparisons. Interestingly, the female subgroup demonstrates a lower MSE (185.77) compared to the male subgroup (219.66), indicating that predictions for women were more tightly clustered around the true values. This suggests the model may generalize slightly better for females, potentially because of less heterogeneity in their cardiac profiles within the dataset. For males, however, the higher MSE and greater variance reflect a broader range of cardiac outcomes, particularly in extreme cases. The "edge cases"—such as very low or very high LVEF values—tend to exhibit higher error, suggesting that the model struggles to handle the most severe or atypical presentations. Yet, around the median LVEF range (50–60%), both genders show strong clustering, reinforcing that the model performs best under normal or near-normal conditions. This pattern supports the notion that while the model achieves high overall accuracy (about 85%), its reduced reliability in extreme cases accounts for the elevated FPR and lower "true" accuracy figure of around 77%. Turning to racial composition, the dataset shows a notable imbalance: 12,851 non-white patients compared to 5,360 white patients. Given that approximately 71% of Americans are white, this constitutes a significant demographic bias. However, this bias may not substantially affect the interpretability of the results, since the physiological mechanisms behind LVEF are not expected to vary dramatically across racial lines. Intriguingly, the white subgroup exhibits more scattering around the median but paradoxically achieves a lower MSE than the non-white subgroup. This

counterintuitive result implies that, while the data for white patients appear more spread out, the model still predicts their LVEF values more accurately on average. Conversely, the minority group data seem more tightly clustered yet yield a higher MSE, suggesting either a higher degree of underlying variability in cardiac function or potential differences in the data collection process. These findings point toward a more complex or possibly minimal influence of race on LVEF predictions, emphasizing that any racial discrepancies observed may stem from sample imbalance or environmental and socioeconomic confounders rather than inherent physiological differences. The analysis of electrocardiographic parameters, including QRS duration, QTc interval, and ventricular rate, adds another crucial layer to this study. Among these, QRS duration demonstrates behavior highly consistent with the previously discussed variables but with a more pronounced effect. Patients in the lower 50% of QRS duration (less than 90 milliseconds) exhibit significantly higher prediction accuracy - approximately 88% - compared to those in the higher QRS group, who show around 84% accuracy. This 4% difference, while modest, reinforces the tendency of the model to perform better on data that fall within normal physiological ranges. The relationship between QRS duration and model accuracy suggests that prolonged QRS intervals, which typically reflect abnormal ventricular conduction, introduce more variability and uncertainty into the predictions. A similar trend emerges with QTc interval measurements. The correlation between QTc and QRS is remarkably strong, with nearly identical MSE values: 159.12 for low QTc (<450 ms) and 159.50 for low QRS, compared to 256.70 and 256.06 for their higher counterparts. This consistency implies that both metrics capture related aspects of cardiac electrical activity that directly influence LVEF estimation. Notably, variance in these measurements is heavily concentrated at the extreme low end, with less than 10% of data points deviating significantly from the optimal prediction line. This concentration suggests that the model is particularly adept at identifying patients with normal or near-normal electrical activity but less effective at handling the irregular patterns associated with prolonged intervals. Consequently, QRS and QTc serve as both strong predictors of LVEF and indicators of where the model may falter - namely, in cases with substantial conduction abnormalities. In contrast, the ventricular rate parameter stands out for its independence from the other variables. Unlike QRS or QTc, the ventricular rate data display a more uniform distribution, lacking the “spiky” concentration patterns observed elsewhere. The median ventricular rate of 82 beats per minute aligns with normal resting values, but its even distribution suggests that heart rate alone may not provide a strong predictive signal for LVEF. This independence reinforces the idea that while heart rate is an important clinical measure, it operates largely orthogonal to the electrical and demographic factors influencing LVEF predictions. In practical terms, this means that while ventricular rate contributes to overall cardiac assessment, its relationship with ejection fraction may be more situational or secondary, depending on the underlying cardiac condition. Bringing these

findings together, a few overarching themes emerge. First, the dataset's age and demographic imbalances introduce measurable biases but do not invalidate the statistical integrity of the results. Second, while the model performs robustly in the midrange of normal cardiac function, its limitations become clear in extreme or atypical cases, where variance and error rates increase substantially. Third, physiological parameters like QRS and QTc durations appear to hold the most predictive power and exhibit the strongest correlations with LVEF, whereas ventricular rate operates independently. Finally, while race and gender introduce observable differences in model performance, these effects appear secondary to the broader physiological and statistical patterns driving LVEF prediction. Ultimately, these findings underscore the nuanced and multidimensional nature of cardiac data analysis. The results suggest that predictive models of LVEF can achieve strong accuracy under normal conditions but require refinement to handle outliers, demographic biases, and complex physiological interactions more effectively. Continued research - especially with more balanced datasets and improved modeling techniques - will be essential for enhancing prediction accuracy and ensuring equitable performance across diverse patient populations. In this way, the study not only sheds light on current model behavior but also lays the groundwork for more comprehensive and inclusive approaches to cardiac health prediction in the future.

Sources:

1. Murkofsky RL, Dangas G, Diamond JA, et al. A prolonged QRS duration on surface electrocardiogram is a specific indicator of left ventricular dysfunction. J Am Coll Cardiol. 1998;32(2):476-482. doi:10.1016/s0735-1097(98)00242-3.
2. Krüger S, Filzmaier K, Graf J, et al. QRS prolongation on surface ECG and brain natriuretic peptide as indicators of left ventricular systolic dysfunction. J Intern Med. 2004;255(2):206-212. doi:10.1046/j.1365-2796.2003.01265.x.
3. Kashani A, Barold SS. Significance of QRS complex duration in patients with heart failure. J Am Coll Cardiol. 2005;46(12):2183-2192. doi:10.1016/j.jacc.2005.01.071.
4. Shenkman HJ, Pampati V, Khandelwal AK, et al. Congestive heart failure and QRS duration: establishing prognosis study. Chest. 2002;122(2):528-534. doi:10.1378/chest.122.2.528.
5. Wang NC, Maggioni AP, Konstam MA, et al. Clinical implications of QRS duration in patients hospitalized with worsening heart failure and reduced left ventricular ejection fraction. JAMA. 2008;299(22):2656-2666. doi:10.1001/jama.299.22.2656.
6. Pai RG, Padmanabhan S. Biological correlates of QT interval and QT dispersion in 2,265 patients with left ventricular ejection fraction ≤40%. J Electrocardiol. 2002;35(3):223-226. doi:10.1054/jelc.2002.33970.

7. Medicare coverage for echocardiogram. AskChapter. Published 2023. Accessed November 16, 2025. **https://askchapter.org/magazine/senior-health-wellness/medicare-resources/does-medicare-cover-an-echocardiogram**
8. Electrocardiogram (EKG/ECG) in office - Cost and procedure. MDsave. Published 2025. Accessed November 16, 2025. **https://www.mdsave.com/procedures/electrocardiogram-ekg-ecg-in-office/d582facf**
9. Elias P, Finer J. EchoNext: A dataset for detecting echocardiogram-confirmed structural heart disease from ECGs.
10. Al Younis SM, Hadjileontiadis LJ, Khandoker AH, et al. Prediction of heart failure patients with distinct left ventricular ejection fraction levels using circadian ECG features and machine learning. PLoS One. 2024;19(5):e0302639. doi:10.1371/journal.pone.0302639.
11. Shirole U, Joshi M. Prediction of left ventricle ejection fraction from HRV in diabetics and cardiac diseased subjects: a mathematical model approach. Medicine in Novel Technology and Devices. 2024;23(Suppl C):100317.
12. Lakshmanan S, Mbanze I. A comparison of cardiovascular imaging practices in Africa, North America, and Europe: two faces of the same coin. Eur Heart J Imaging Methods Pract. 2023;1(1):qyad005. doi:10.1093/ehjimp/qyad005.
13. Leese PJ, Viera AJ, Hinderliter AL, Stearns SC. Cost-effectiveness of electrocardiography vs. electrocardiography plus limited echocardiography to diagnose LVH in young, newly identified hypertensives. Am J Hypertens. 2010;23(6):592-598. doi:10.1038/ajh.2010.42
14. Goldberger A, Amaral L, Glass L, Hausdorff J, Ivanov PC, Mark R, et al. PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals. Circulation. 2000;101(23):e215-e220. RRID:SCR_007345.

Acknowledgements:

1. Creators of the Echo Next Data base
2. Dr. Abhinav Chandra