CA2 - Supervised machine learning classification pipeline - applied to medical data

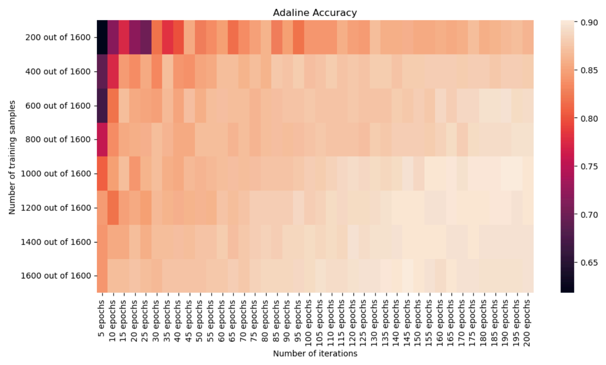
Group 56: Vetle Rosseland, Mads Teien, Eirik Årdal

Important information

- Do **not** use scikit-learn (`sklearn`) or any other high-level machine learning library for this CA
- Explain your code and reasoning in markdown cells or code comments
- Label all graphs and charts if applicable
- If you use code from the internet, make sure to reference it and explain it in your own words
- If you use additional function arguments, make sure to explain them in your own words
- Use the classes `Perceptron` , `Adaline` and `Logistic Regression` from the library `mlxtend` as classifiers (`from mlxtend.classifier import Perceptron, Adaline, LogisticRegression`). *Always* use the argument `minibatches=1` when instantiating an `Adaline` or `LogisticRegression` object. This makes the model use the gradient descent algorithm for training. Always use the `random_seed=42` argument when instantiating the classifiers. This will make your results reproducible.
- You can use any plotting library you want (e.g. `matplotlib` , `seaborn` , `plotly` , etc.)
- Use explanatory variable names (e.g. `X_train` and `X_train_scaled` for the training data before and after scaling, respectively)
- The dataset is provided in the file `fetal_health.csv` in the `assets` folder

Additional clues

- Use the `pandas` library for initial data inspection and preprocessing
- Before training the classifiers, convert the data to raw `numpy` arrays
- For Part IV, you are aiming to create a plot that looks similar to this:



Additional information

- Feel free to create additional code or markdown cells if you think it will help you explain your reasoning or structure your code (you don't have to).

Part I: Data loading and data exploration

Import necessary libraries/modules:

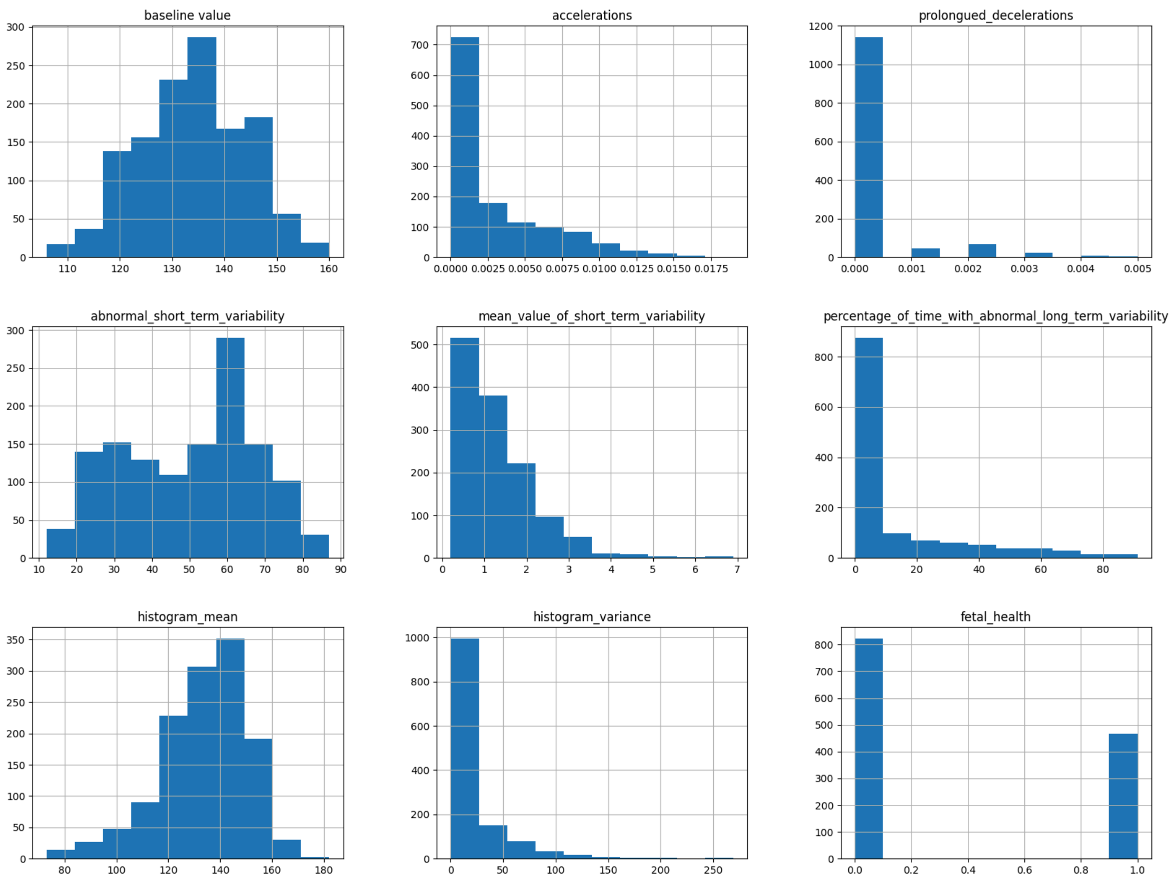
In [3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from mlxtend.classifier import Perceptron, Adaline, LogisticRegression
```

Loading and exploring data

1. Load the dataset `fetal_health.csv` with `pandas`. Use the first column as the row index.
2. Check for missing data, report on your finding and remove samples with missing data, if you find any.
3. Display the raw data with appropriate plots/outputs and inspect it. Describe the distributions of the values of feature `"baseline value"`, `"accelerations"`, and the target variable `"fetal_health"`.
4. Will it be beneficial to scale the data? Why or why not? Scaling will significantly improve model performance, training efficiency, and result interpretability for this medical classification task.
5. Is the data linearly separable using a combination of any two pairs of features? Can we expect an accuracy close to 100% from a linear classifier? No we shouldnt expect 100%. Something in the 70-85% range is more the be expected

In [4]:
```
# Load the iris dataset
df = pd.read_csv('assets/fetal_health.csv', index_col=0)
df.head()

df.hist(figsize=(20, 15))
plt.show()
```



In [5]:
```python
missing_values = df.isna().sum()
print("Missing values per column:")
print(missing_values)

total_missing = df.isna().sum().sum()
print(f"\nTotal missing values in the dataset: {total_missing}")
```

```
Missing values per column:
baseline value                                            0
accelerations                                             0
prolongued_decelerations                                  0
abnormal_short_term_variability                           0
mean_value_of_short_term_variability                      0
percentage_of_time_with_abnormal_long_term_variability    0
histogram_mean                                            0
histogram_variance                                        0
fetal_health                                              0
dtype: int64

Total missing values in the dataset: 0
```

Part II: Train/Test Split

Divide your dataset into training and testing subsets. Follow these steps to create the split:

1. **Divide the dataset into two data sets, each data set only contains samples of either class 0 or class 1:**

- Create a DataFrame `df_0` containing all data with `"fetal_health"` equal to 0.
- Create a DataFrame `df_1` containing all data with `"fetal_health"` equal to 1.

2. **Split into training and test set by randomly sampling entries from the data frames:**

- Create a DataFrame `df_0_train` containing by sampling `75%` of the entries from `df_0` (use the `sample` method of the data frame, fix the `random_state` to `42`).
- Create a DataFrame `df_1_train` using the same approach with `df_1`.
- Create a DataFrame `df_0_test` containing the remaining entries of `df_0` (use `df_0.drop(df_0_train.index)` to drop all entries except the previously extracted ones).
- Create a DataFrame `df_1_test` using the same approach with `df_1`.

3. **Merge the datasets split by classes back together:**

- Create a DataFrame `df_train` containing all entries from `df_0_train` and `df_1_train`. (Hint: use the `concat` method you know from CA1)
- Create a DataFrame `df_test` containing all entries from the two test sets.

4. **Create the following data frames from these splits:**

- `X_train`: Contains all columns of `df_train` except for the target feature `"fetal_health"`
- `X_test`: Contains all columns of `df_test` except for the target feature `"fetal_health"`
- `y_train`: Contains only the target feature `"fetal_health"` for all samples in the training set
- `y_test`: Contains only the target feature `"fetal_health"` for all samples in the test set

5. **Check that your sets have the expected sizes/shape by printing number of rows and colums ("shape") of the data sets.**

- (Sanity check: there should be 8 features, almost 1000 samples in the training set and slightly more than 300 samples in the test set.)

6. **Explain the purpose of this slightly complicated procedure. Why did we first split into the two classes? Why did we then split into a training and a testing set?**

7. **What is the share (in percent) of samples with class 0 label in test and training set, and in the intial data set?**

In [6]:
```python
df_0 = df[df['fetal_health'] == 0]
df_1 = df[df['fetal_health'] == 1]
```

In [7]:
```python
df_0_train = df_0.sample(frac=0.75, random_state=42)
df_0_test = df_0.drop(df_0_train.index)
df_1_train = df_1.sample(frac=0.75, random_state=42)
df_1_test = df_1.drop(df_1_train.index)

df_train = pd.concat([df_0_train, df_1_train])
```

```
df_test = pd.concat([df_0_test, df_1_test])

X_train = df_train.drop('fetal_health', axis=1)
X_test = df_test.drop('fetal_health', axis=1)
y_train = df_train['fetal_health']
y_test = df_test['fetal_health']
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(967, 8) (323, 8) (967,) (323,)
```

6. The purpose of this procedure for splitting the data into training and testing sets is to ensure that both sets have a representative distribution of each class. This method is known as stratified sampling. This procedure helps in preserving the initial class ratios, which is particularly important for classification problems where class imbalance might skew the model’s performance.

In [8]:
```
class_0_share_train = (y_train == 0).mean() * 100
class_0_share_test = (y_test == 0).mean() * 100
class_0_share_initial = (df['fetal_health'] == 0).mean() * 100
print(f"Share of samples with class 0 label in the sets: training:{class_0_share
```

```
Share of samples with class 0 label in the sets: training:63.8%, testing: 63.8%,
initial:63.8%
```

Convert data to numpy arrays and shuffle the training data

Many machine learning models (including those you will work with later in the assignment) will not accept DataFrames as input. Instead, they will only work if you pass numpy arrays containing the data. Here, we convert the DataFrames `X_train`, `X_test`, `y_train`, and `y_test` to numpy arrays `X_train`, `X_test`, `y_train`, and `y_test`.

Moreover we shuffle the training data. This is important because the training data is currently ordered by class. In Part IV, we use the first n samples from the training set to train the classifiers. If we did not shuffle the data, the classifiers would only be trained on samples of class 0.

Nothing to be done here, just execute the cell.

In [9]:
```
# convert to numpy arrays
X_train = X_train.to_numpy()
X_test = X_test.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()

# shuffle training data
np.random.seed(42) # for reproducibility
shuffle_index = np.random.permutation(len(X_train)) # generate random indices
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index] # shuffle data
```

Part III: Scaling the data

1. Standardize the training *and* test data so that each feature has a mean of 0 and a standard deviation of 1.
2. Check that the scaling was successful
   - by printing the mean and standard deviation of each feature in the scaled training set
   - by putting the scaled training set into a DataFrame and make a violin plot of the data

Hint: use the `axis` argument to calculate mean and standard deviation column-wise.

Important: Avoid data leakage!

More hints:

1. For each column, subtract the mean (μ) of each column from each value in the column
2. Divide the result by the standard deviation (σ) of the column

(You saw how to do both operations in the lecture. If you don't remember, you can look it up in Canvas files.)

Mathematically (in case this is useful for you), this transformation can be represented for each column as follows:

$$X_{\text{scaled}} = \frac{(X - \mu)}{\sigma}$$

where:

- (X_{scaled}) are the new, transformed column values (a column-vector)
- (X) is the original values
- (μ) is the mean of the column
- (σ) is the standard deviation of the column

In [10]:
```
X_train_scaled = (X_train - X_train.mean(axis=0)) / X_train.std(axis=0)
X_test_scaled = (X_test - X_train.mean(axis=0)) / X_train.std(axis=0)
```

In [11]:
```
# Checking the mean and standard deviation of the scaled features
print(X_train_scaled.mean(axis=0))
print(X_train_scaled.std(axis=0))
```

```
[-1.31803106e-16  4.56925087e-15 -2.96097744e-16  1.33869705e-16
 -2.12543989e-17 -2.86453614e-16 -2.93342278e-16 -7.18717284e-17]
[1. 1. 1. 1. 1. 1. 1. 1.]
```

Part IV: Training and evaluation with different dataset sizes and training times

Often, a larger dataset size will yield better model performance. (As we will learn later, this usually prevents overfitting and increases the generalization capability of the trained model.) However, collecting data is usually rather expensive.

In this part of the exercise, you will investigate

- how the model performance changes with varying dataset size
- how the model performance changes with varying numbers of epochs/iterations of the optimizer/solver (increasing training time).

For this task (Part IV), use the `Adaline`, `Perceptron`, and `LogisticRegression` classifier from the `mlxtend` library. All use the gradient descent (GD) algorithm for training.

Important: Use a learning rate of `1e-4` (`0.0001`) for all classifiers, and use the argument `minibatches=1` when initializing `Adaline` and `LogisticRegression` classifier (this will make sure it uses GD). For all three classifiers, pass `random_seed=42` when initializing the classifier to ensure reproducibility of the results.

Model training

Train the model models using progressively larger subsets of your dataset, specifically: first 50 rows, first 100 rows, first 150 rows, ..., first 650 rows, first 700 rows (in total 14 different variants).

For each number of rows train the model with progressively larger number of epochs: 2, 7, 12, 17, ..., 87, 92, 97 (in total 20 different model variants).

The resulting $14 \times 20 = 280$ models obtained from the different combinations of subsets and number of epochs. An output of the training process could look like this:

```
Model (1) Train a model with first 50 rows of data for 2 epochs
Model (2) Train a model with first 50 rows of data for 7 epochs
Model (3) Train a model with first 50 rows of data for 12 epochs
...
Model (21) Train a model with first 100 rows of data for 2
epochs
Model (22) Train a model with first 100 rows of data for 7
epochs
...
Model (279) Train a model with first 700 rows of data for 92
epochs
Model (280) Train a model with first 700 rows of data for 97
epochs
```

Model evaluation

For each of the 280 models, calculate the **accuracy on the test set** (do **not** use the score method but compute accuracy yourself). Store the results in the provided 2D numpy array (it has 14 rows and 20 columns). The rows of the array correspond to the different dataset sizes, and the columns correspond to the different numbers of epochs.

Tasks

1. Train the 280 Adaline classifiers as mentioned above and calculate the accuracy for each of the 280 variants.
2. Generalize your code so that is doing the same procedure for all three classifiers: `Perceptron`, `Adaline`, and `LogisticRegression` after each other. Store the result for all classifiers. You can for example use an array of shape $3 \times 14 \times 20$ to store the accuracies of the three classifiers.

Note that executing the cells will take some time (but on most systems it should not be more than 5 minutes).

In [12]:
```python
# Train and evaluate all model variants
accuracies = np.zeros((3, 14, 20))  # 3 models, 14 dataset sizes, 20 epochs
dataset_sizes = np.arange(50, 701, 50)  # 14 dataset sizes
epochs = np.arange(2, 98, 5)  # 20 epochs
classifiers = [Perceptron, Adaline, LogisticRegression]

for i, Classifier in enumerate(classifiers):
    print(f"Training {Classifier.__name__} models...")
    for j, size in enumerate(dataset_sizes):
        X_train_subset = X_train_scaled[:size]
        y_train_subset = y_train[:size]
        for k, epoch in enumerate(epochs):
            if Classifier == Perceptron:
                model_instance = Classifier(epochs=epoch, eta=0.0001, random_see
            else:
                model_instance = Classifier(epochs=epoch, eta=0.0001, random_see

            model_instance.fit(X_train_subset, y_train_subset)
            y_pred = model_instance.predict(X_test_scaled)
            accuracies[i, j, k] = (y_pred == y_test).mean()

        # Print progress for each dataset size completion
        print(f"Model: {Classifier.__name__}, Dataset size: {size}, Epochs: {epc
```

```
Training Perceptron models...
Model: Perceptron, Dataset size: 50, Epochs: 97, Accuracy: 0.89
Model: Perceptron, Dataset size: 100, Epochs: 97, Accuracy: 0.87
Model: Perceptron, Dataset size: 150, Epochs: 97, Accuracy: 0.86
Model: Perceptron, Dataset size: 200, Epochs: 97, Accuracy: 0.89
```

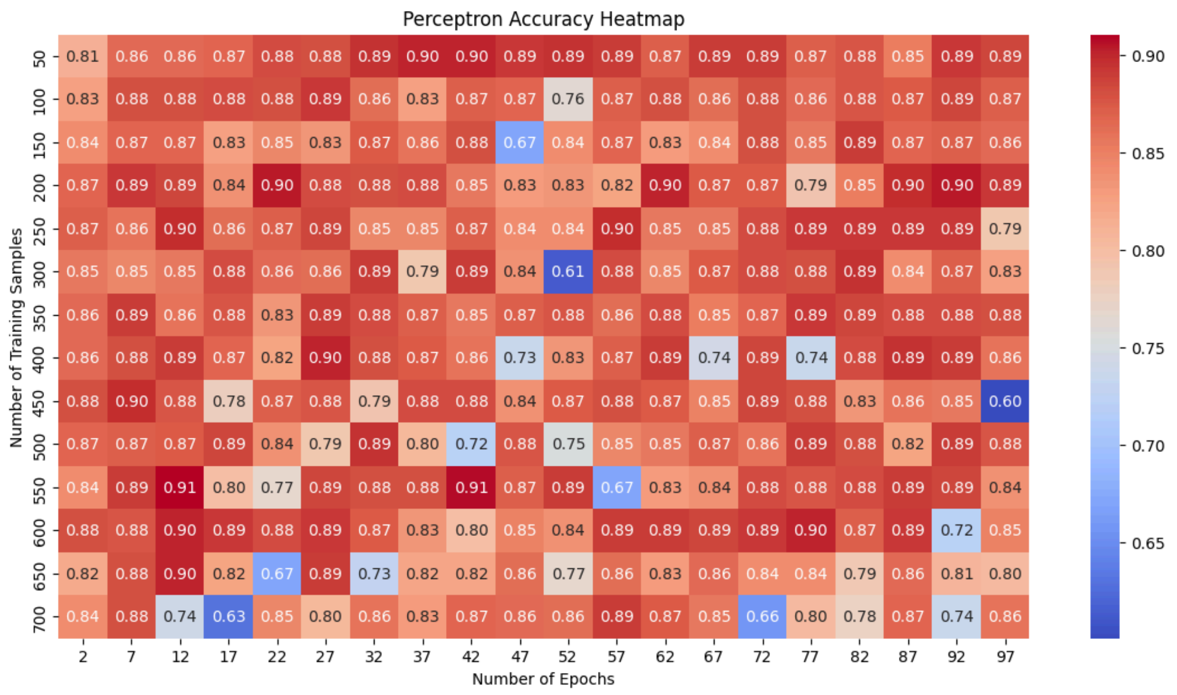
```
Model: Perceptron, Dataset size: 250, Epochs: 97, Accuracy: 0.79
Model: Perceptron, Dataset size: 300, Epochs: 97, Accuracy: 0.83
Model: Perceptron, Dataset size: 350, Epochs: 97, Accuracy: 0.88
Model: Perceptron, Dataset size: 400, Epochs: 97, Accuracy: 0.86
Model: Perceptron, Dataset size: 450, Epochs: 97, Accuracy: 0.60
Model: Perceptron, Dataset size: 500, Epochs: 97, Accuracy: 0.88
Model: Perceptron, Dataset size: 550, Epochs: 97, Accuracy: 0.84
Model: Perceptron, Dataset size: 600, Epochs: 97, Accuracy: 0.85
Model: Perceptron, Dataset size: 650, Epochs: 97, Accuracy: 0.80
Model: Perceptron, Dataset size: 700, Epochs: 97, Accuracy: 0.86
Training Adaline models...
Model: Adaline, Dataset size: 50, Epochs: 97, Accuracy: 0.87
Model: Adaline, Dataset size: 100, Epochs: 97, Accuracy: 0.88
Model: Adaline, Dataset size: 150, Epochs: 97, Accuracy: 0.87
Model: Adaline, Dataset size: 200, Epochs: 97, Accuracy: 0.87
Model: Adaline, Dataset size: 250, Epochs: 97, Accuracy: 0.87
Model: Adaline, Dataset size: 300, Epochs: 97, Accuracy: 0.87
Model: Adaline, Dataset size: 350, Epochs: 97, Accuracy: 0.88
Model: Adaline, Dataset size: 400, Epochs: 97, Accuracy: 0.87
Model: Adaline, Dataset size: 450, Epochs: 97, Accuracy: 0.89
Model: Adaline, Dataset size: 500, Epochs: 97, Accuracy: 0.89
Model: Adaline, Dataset size: 550, Epochs: 97, Accuracy: 0.89
Model: Adaline, Dataset size: 600, Epochs: 97, Accuracy: 0.89
Model: Adaline, Dataset size: 650, Epochs: 97, Accuracy: 0.89
Model: Adaline, Dataset size: 700, Epochs: 97, Accuracy: 0.89
Training LogisticRegression models...
Model: LogisticRegression, Dataset size: 50, Epochs: 97, Accuracy: 0.86
Model: LogisticRegression, Dataset size: 100, Epochs: 97, Accuracy: 0.88
Model: LogisticRegression, Dataset size: 150, Epochs: 97, Accuracy: 0.88
Model: LogisticRegression, Dataset size: 200, Epochs: 97, Accuracy: 0.87
Model: LogisticRegression, Dataset size: 250, Epochs: 97, Accuracy: 0.87
Model: LogisticRegression, Dataset size: 300, Epochs: 97, Accuracy: 0.88
Model: LogisticRegression, Dataset size: 350, Epochs: 97, Accuracy: 0.87
Model: LogisticRegression, Dataset size: 400, Epochs: 97, Accuracy: 0.87
Model: LogisticRegression, Dataset size: 450, Epochs: 97, Accuracy: 0.88
Model: LogisticRegression, Dataset size: 500, Epochs: 97, Accuracy: 0.88
Model: LogisticRegression, Dataset size: 550, Epochs: 97, Accuracy: 0.87
Model: LogisticRegression, Dataset size: 600, Epochs: 97, Accuracy: 0.88
Model: LogisticRegression, Dataset size: 650, Epochs: 97, Accuracy: 0.88
Model: LogisticRegression, Dataset size: 700, Epochs: 97, Accuracy: 0.88
```

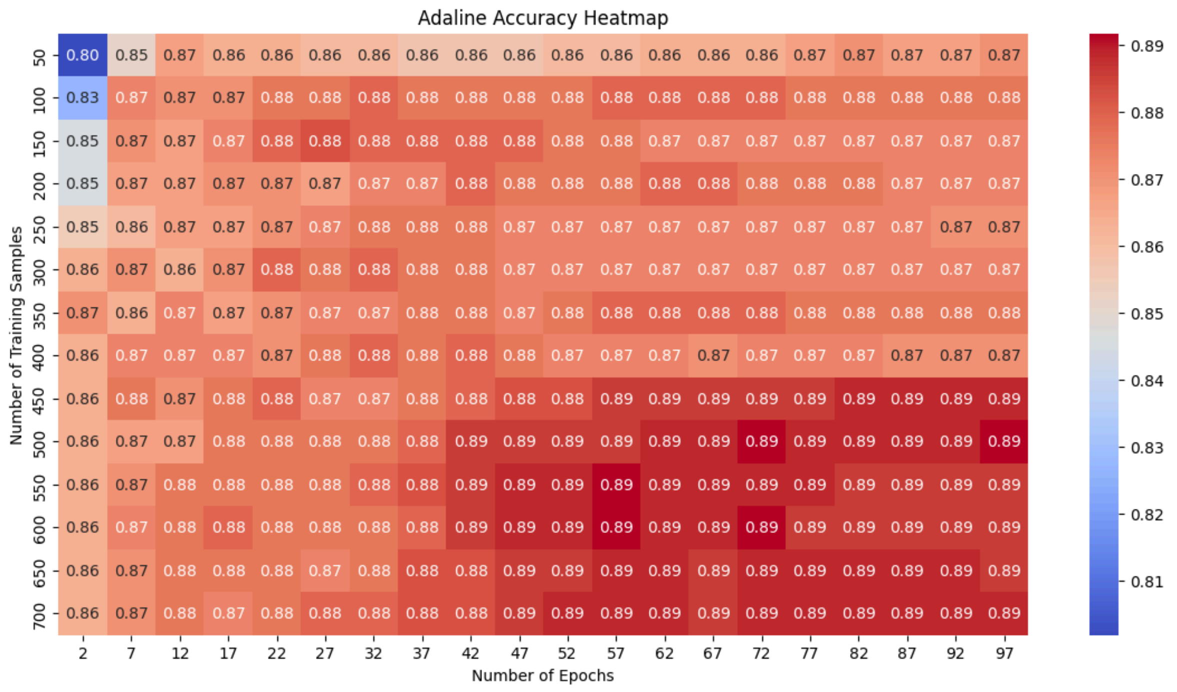
Performance visualization

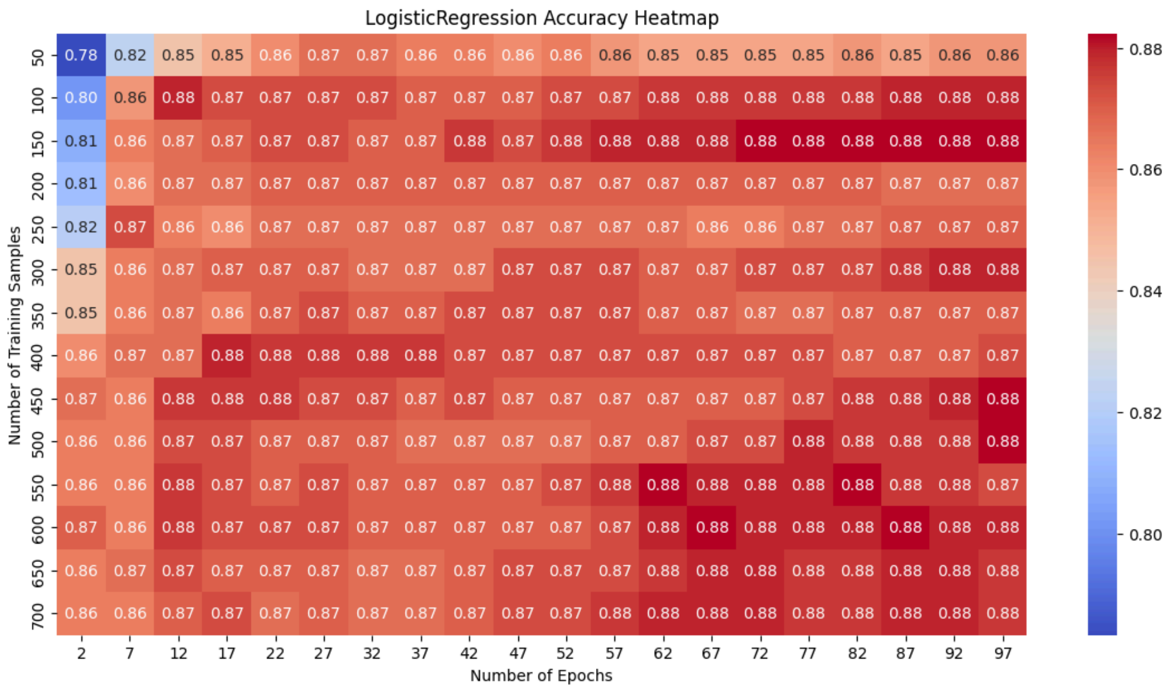
Plot the performance measure for all classifiers (accuracy on the test set; use the result array from above) of all the 280 variants for each classifier in a total of three heatmaps using, for example `seaborn` or `matplotlib` directly.

The color should represent the accuracy on the test set, and the x and y axes should represent the number of epochs and the dataset size, respectively. Which one is x and which one is y is up to you to decide. Look in the example output at the top of the assignment for inspiration for how the plot could look like and how it could be labeled nicely. (But use the correct numbers corresponding to your dataset sizes and number of epochs.)

In [13]:
```python
classifier_names = [classifier.__name__ for classifier in classifiers]
for i, name in enumerate(classifier_names):
    plt.figure(figsize=(14, 7))
    sns.heatmap(accuracies[i], annot=True, fmt=".2f", cmap='coolwarm',xticklabel
    plt.title(f'{name} Accuracy Heatmap')
    plt.xlabel('Number of Epochs')
    plt.ylabel('Number of Training Samples')
    plt.show()
```







Part V: Some more plotting

For the following cell to execute you need to have the variable `X_test_scaled` with all samples of the test set and the variable `y_test` with the corresponding labels. Complete at least up until Part III. Executing the cell will plot something.

1. Add code comments explaining what the lines are doing
2. What is the purpose of the plot?
3. Describe all components of the subplot and then comment in general on the entire plot. What does it show? What does it not show?

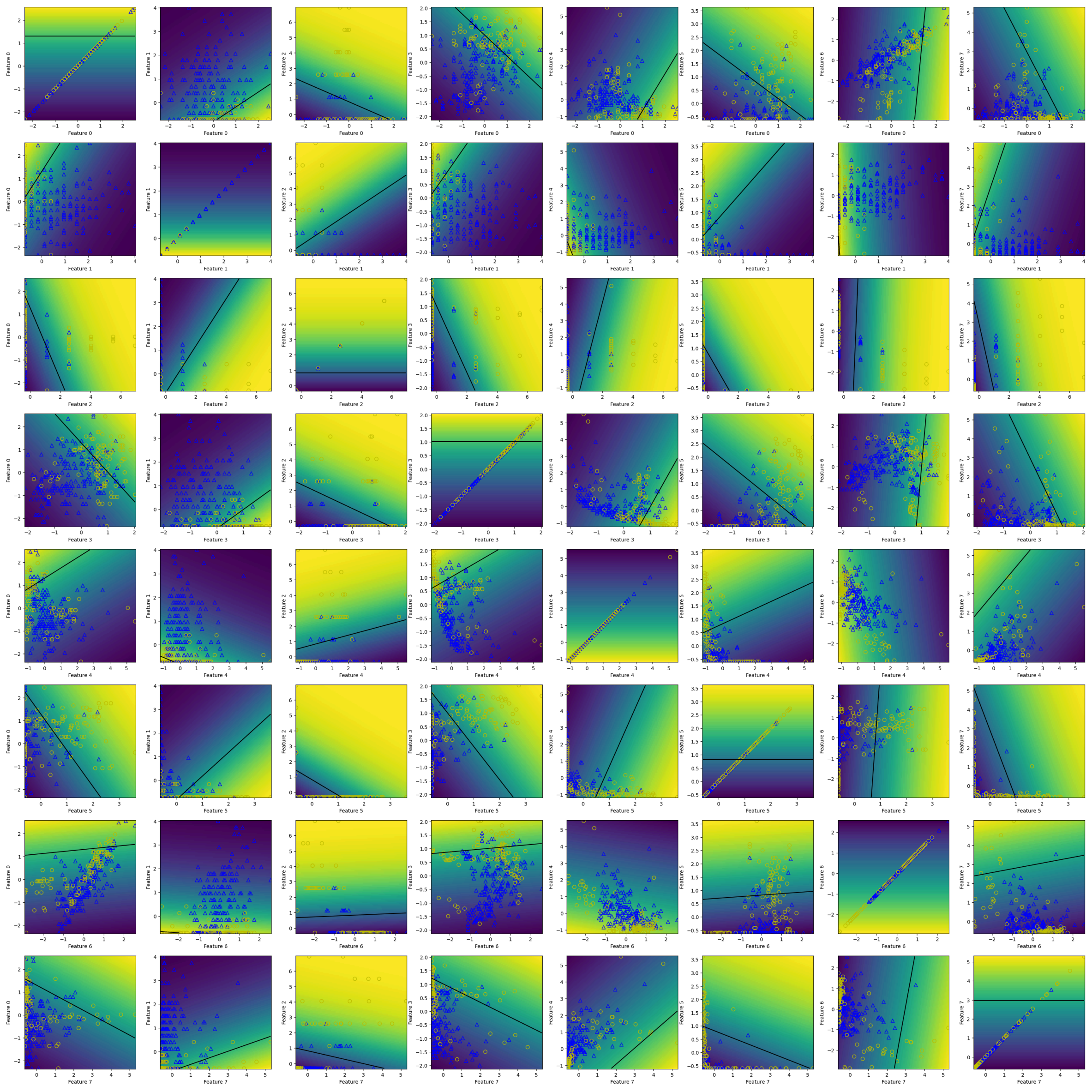
In [14]:
```python
# Train and a logistic regression model with 300 epochs and learning rate 0.0001
clf = LogisticRegression(eta = 0.0001, epochs = 300, minibatches=1, random_seed=
clf.fit(X_test_scaled, y_test)
# Fit the classifier to the scaled test data
# Create a figure with a grid of subplots of 8x8 size, with each subplot's size
fig, axes = plt.subplots(8, 8, figsize=(30, 30))

# Loop through all possible pairs of features
for i in range(0, 8):
    for j in range(0, 8):
        feature_1 = i
        feature_2 = j
        ax = axes[i, j]

        # Set the title of the subplot to the pair of features
        ax.set_xlabel(f"Feature {feature_1}")
        ax.set_ylabel(f"Feature {feature_2}")

        mins = X_test_scaled.min(axis=0)
        maxs = X_test_scaled.max(axis=0)
        # Create a grid of 100x100 points for the two features
        x0 = np.linspace(mins[feature_1], maxs[feature_1], 100)
        x1 = np.linspace(mins[feature_2], maxs[feature_2], 100)
```

```
            # Create a meshgrid from the two features
            X0, X1 = np.meshgrid(x0, x1)
            # Create a 2D array of the two features
            X_two_features = np.c_[X0.ravel(), X1.ravel()]
            X_plot = np.zeros(shape=(X_two_features.shape[0], X_test_scaled.shape[1]
            # Set the two features in the 2D array to the two features we are plotti
            X_plot[:, feature_1] = X_two_features[:, 0]
            X_plot[:, feature_2] = X_two_features[:, 1]
            # Predict the probabilities for each class for each point in the grid
            y_pred = clf.predict_proba(X_plot)
            # Reshape the predictions to a 2D grid
            Z = y_pred.reshape(X0.shape)
            # Plot the decision boundary
            ax.pcolor(X0, X1, Z)
            ax.contour(X0, X1, Z, levels=[0.5], colors='k')
            # Plot the test data points
            ax.scatter(X_test_scaled[y_test == 0, feature_1], X_test_scaled[y_test =
            ax.scatter(X_test_scaled[y_test == 1, feature_1], X_test_scaled[y_test =
    # Adjust the layout of the subplots
    fig.tight_layout()
    plt.show()
```



Part VI: Additional discussion

Part I:

1. What kind of plots did you use to visualize the raw data, and why did you choose these types of plots?

We used histograms, as it displays the values seperatly so we can understand the features that we work with

Part II:

1. What happens if we don't shuffle the training data before training the classifiers like in Part IV? The classifier might learn the patterns of the first few classes it sees and might not generalize well. For algorithms that use gradient descent, starting with data in a particular order can bias the gradient updates, especially in the early stages of training.
2. How could you do the same train/test split (Point 1.-4.) using scikit-learn? We could start by using the train_test_split function Preprocess the Data: Apply feature scaling using built-in scalers like StandardScaler. This would normalize the data, ensuring that all features contribute equally to the model's predictions. Train Classifiers: Use scikit-learn's classifier implementations, such as Perceptron, Logisti- cRegression, and SGDClassifier (the latter can mimic an ADALINE). These classifiers come with methods for easy fitting to the training data. Evaluate Models: Utilize scikit-learn's metrics, such as accuracy_score and confusion_matrix, to evaluate the performance of your classifiers. These functions provide a straightforward way to assess accuracy and understand misclassifications. Visualize Results: Employ scikit-learn's plotting capabilities or other libraries like Matplotlib to create visualizations like confusion matrices, which can help in interpreting the performance of your models.

Part IV:

1. How does increasing the dataset size affect the performance of the logistic regression model? Provide a summary of your findings. In general the accuracy seems to increase with the size of the dataset, upto a certain size atleast. It's most noticable with fewer epochs, like with 2 epochs you can clearly see the accuracy increasing with the size, whilst with with a higher epoch there is barely difference between the sizes
2. Describe the relationship between the number of epochs and model accuracy Across all models, we can observe that there is a plateau effect; after a certain number of 18 epochs, the improvements in accuracy diminish, indicating that additional training beyond this point does not significantly benefit the model. Logistic Regression: The accuracy appears relatively stable across different numbers of epochs, suggesting that the logistic regression model converges to its optimal performance quickly and remains stable with additional training. Adaline: There seems to be a trend where accuracy improves slightly as the number of epochs increases, especially noticeable in the mid-range of training samples. This indicates that the Adaline model may benefit from more iterations over the dataset. Perceptron: The perceptron model shows some volatility in accuracy as the number of epochs increases. In some cases,

more epochs lead to higher accuracy, but there are instances where accuracy decreases, which could suggest overfitting or instability in the learning process for this model.

3. Which classifier is much slower to train and why do you think that is? Perceptron was much slower. This is because the weights get updated after each misclassified sample which means it updates much more than the other two.
4. One classifier shows strong fluctuations in accuracy for different dataset sizes and number of epochs. Which one is it and why do you think this happens? Perceptron. The perceptron algorithm is designed to converge only if the data is linearly separable. If the data can’t be separated by a linear boundary, the algorithm might not converge, leading to fluctuations in accuracy as it continues to seek a boundary that doesn’t exist.