Assignment 3 - Supervised Learning: model training and evaluation

Nick Carroll

Netid: nc230

Note: this assignment falls under collaboration Mode 2: Individual Assignment – Collaboration Permitted. Please refer to the syllabus for additional information.

Instructions for all assignments can be found here, and is also linked to from the course syllabus.

Total points in the assignment add up to 90; an additional 10 points are allocated to presentation quality.

Learning Objectives:

This assignment will provide structured practice to help enable you to...

1. Understand the primary workflow in machine learning: (1) identifying a hypothesis function set of models, (2) determining a loss/cost/error/objective function to minimize, and (3) minimizing that function through gradient descent
2. Understand the inner workings of logistic regression and how linear models for classification can be developed.
3. Gain practice in implementing machine learning algorithms from the most basic building blocks to understand the math and programming behind them to achieve practical proficiency with the techniques
4. Implement batch gradient descent and become familiar with how that technique is used and its dependence on the choice of learning rate
5. Evaluate supervised learning algorithm performance through ROC curves and using cross validation
6. Apply regularization to linear models to improve model generalization performance

1

Classification using logistic regression: build it from the ground up

[60 points]

This exercise will walk you through the full life-cycle of a supervised machine learning classification problem. Classification problem consists of two features/predictors (e.g. petal width and petal length) and your goal is to predict one of two possible classes (class 0 or class 1). You will build, train, and evaluate the performance of a logistic regression classifier on the data provided. Before you begin any modeling, you'll load and explore your data in Part I to familiarize yourself with it - and check for any missing or erroneous data. Then, in Part II, we will review an appropriate hypothesis set of functions to fit to the data: in this case, logistic regression. In Part III, we will derive an appropriate cost function for the data (spoiler alert: it's cross-entropy) as well as the gradient descent update equation that will allow you to optimize that cost function to identify the parameters that minimize the cost for the training data. In Part IV, all the pieces come together and you will implement your logistic regression model class including methods for fitting the data using gradient descent. Using that model you'll test it out and plot learning curves to verify the model learns as you train it and to identify and appropriate learning rate hyperparameter. Lastly, in Part V you will apply the model you designed, implemented, and verified to your actual data and evaluate and visualize its generalization performance as compared to a KNN algorithm. **When complete, you will have accomplished learning objectives 1-5 above!**

I. Load, prepare, and plot your data

You are given some data for which you are tasked with constructing a classifier. The first step when facing any machine learning project: look at your data!

(a) Load the data.

- In the data folder in the same directory of this notebook, you'll find the data in `A3_Q1_data.csv`. This file contains the binary class labels, y, and the features x_1 and x_2.
- Divide your data into a training and testing set where the test set accounts for 30 percent of the data and the training set the remaining 70 percent.
- Plot the training data by class.
- Comment on the data: do the data appear separable? May logistic regression be a good choice for these data? Why or why not?

(b) Do the data require any preprocessing due to missing values, scale differences (e.g. different ranges of values), etc.? If so, how did you handle these issues?

Next, we walk through our key steps for model fitting: choose a hypothesis set of models to train (in this case, logistic regression); identify a cost function to measure the model fit to our training data; optimize model parameters to minimize cost (in this case using gradient descent). Once we've completed model fitting, we will evaluate the performance of our model and compare performance to another approach (a KNN classifier).

II. Stating the hypothesis set of models to evaluate (we'll use logistic regression)

Given that our data consists of two features, our logistic regression problem will be applied to a two-dimensional feature space. Recall that our logistic regression model is:

$$f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{x}_i)$$

where the sigmoid function is defined as $\sigma(x) = \dfrac{e^x}{1+e^x} = \dfrac{1}{1+e^{-x}}$. Also, since this is a two-dimensional problem, we define $\mathbf{w}^\top \mathbf{x}_i = w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2}$ and here, $\mathbf{x}_i = [x_{i,0}, x_{i,1}, x_{i,2}]^\top$, and $x_{i,0} \triangleq 1$

Remember from class that we interpret our logistic regression classifier output (or confidence score) as the conditional probability that the target variable for a given sample y_i is from class "1", given the observed features, $\mathbf{x}_i$. For one sample, $(y_i, \mathbf{x}_i)$, this is given as:

$$P(Y = 1 | X = \mathbf{x}_i) = f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{x}_i)$$

In the context of maximizing the likelihood of our parameters given the data, we define this to be the likelihood function $L(\mathbf{w}|y_i, \mathbf{x}_i)$, corresponding to one sample observation from the training dataset.

Aside: the careful reader will recognize this expression looks different from when we talk about the likelihood of our data given the true class label, typically expressed as $P(x|y)$, or the posterior probability of a class label given our data, typically expressed as $P(y|x)$. In the context of training a logistic regression model, the likelihood we are interested in is the likelihood function of our logistic regression parameters, $\mathbf{w}$. It's our goal to use this to choose the parameters to maximize the likelihood function.

No output is required for this section - just read and use this information in the later sections.

III. Find the cost function that we can use to choose the model parameters, $\mathbf{w}$, that best fit the training data.

(c) What is the likelihood function that corresponds to all the N samples in our training dataset that we will wish to maximize? Unlike the likelihood function written above which gives the likelihood function for a *single training data pair* $(y_i, \mathbf{x}_i)$, this question asks for the likelihood function for the *entire training dataset* $\{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_N, \mathbf{x}_N)\}$.

(d) Since a logarithm is a monotonic function, maximizing the $f(x)$ is equivalent to maximizing $\ln[f(x)]$. Express the likelihood from the last question as a cost function of the model parameters, $C(\mathbf{w})$; that is the negative of the logarithm of the likelihood. Express this cost as an average cost per sample (i.e. divide your final value by N), and use this quantity going forward as the cost function to optimize.

(e) Calculate the gradient of the cost function with respect to the model parameters $\nabla_{\mathbf{w}} C(\mathbf{w})$. Express this in terms of the partial derivatives of the cost function with respect to each of the parameters, e.g. $\nabla_{\mathbf{w}} C(\mathbf{w}) = \left[\dfrac{\partial C}{\partial w_0}, \dfrac{\partial C}{\partial w_1}, \dfrac{\partial C}{\partial w_2}\right]$.

To simplify notation, please use $\mathbf{w}^\top \mathbf{x}$ instead of writing out $w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2}$ when it appears each time (where $x_{i,0} = 1$ for all i). You are also welcome to use $\sigma()$ to represent the sigmoid function. Lastly, this will be a function the features, $x_{i,j}$ (with the first index in the subscript representing the observation and the second the feature; targets, y_i; and the logistic regression model parameters, w_j.

(f) Write out the gradient descent update equation. This should clearly express how to update each weight from one step in gradient descent $w_j^{(k)}$ to the next $w_j^{(k+1)}$. There should be one equation for each model logistic regression model parameter (or you can represent it in vectorized form). Assume that η represents the learning rate.

IV. Implement gradient descent and your logistic regression algorithm

(g) Implement your logistic regression model.

- You are provided with a template, below, for a class with key methods to help with your model development. It is modeled on the Scikit-Learn convention. For this, you only need to create a version of logistic regression for the case of two feature variables (i.e. two predictors).
- Create a method called `sigmoid` that calculates the sigmoid function
- Create a method called `cost` that computes the cost function $C(\mathbf{w})$ for a given dataset and corresponding class labels. This should be the **average cost** (make sure your total cost is divided by your number of samples in the dataset).
- Create a method called `gradient_descent` to run **one step** of gradient descent on your training data. We'll refer to this as "batch" gradient descent since it takes into account the gradient based on all our data at each iteration of the algorithm.
- Create a method called `fit` that fits the model to the data (i.e. sets the model parameters to minimize cost) using your `gradient_descent` method. In doing this we'll need to make some assumptions about the following:
  - Weight initialization. What should you initialize the model parameters to? For this, randomly initialize the weights to a different values between 0 and 1.
  - Learning rate. How slow/fast should the algorithm step towards the minimum? This you will vary in a later part of this problem.
  - Stopping criteria. When should the algorithm be finished searching for the optimum? There are two stopping criteria: small changes in the gradient descent step size and a maximum number of iterations. The first is whether there was a sufficiently small change in the gradient; this is evaluated as whether the magnitude of the step that the gradient descent algorithm takes changes by less than 10^{-6} between iterations. Since we have a weight vector, we can compute the change in the weight by evaluating the L_2 norm (Euclidean norm) of the change in the vector between iterations. From our gradient descent update equation we know that mathematically this is $||-\eta \nabla_{\mathbf{w}} C(\mathbf{w})||$. The second criterion is met if a maximum number of iterations has been reach (5,000 in this case, to prevent infinite loops from poor choices of learning rates).
  - Design your approach so that at each step in the gradient descent algorithm you evaluate the cost function for both the training and the test data for each new value for the model weights. You should be able to plot cost vs gradient descent iteration for both the training and the test data. This will allow you to plot "learning curves" that can be informative for how the model training process is proceeding.
- Create a method called `predict_proba` that predicts confidence scores (that can be thresholded into the predictions of the `predict` method.
- Create a method called `predict` that makes predictions based on the trained model, selecting the most probable class, given the data, as the prediction, that is class that yields the larger $P(y|\mathbf{x})$.
- (Optional, but recommended) Create a method called `learning_curve` that produces the cost function values that correspond to each step from a previously run gradient descent operation.
- (Optional, but recommended) Create a method called `prepare_x` which appends a column of ones as the first feature of the dataset $\mathbf{X}$ to account for the bias term ($x_{i,1} = 1$).

This structure is strongly encouraged; however, you're welcome to adjust this to your needs (adding helper methods, modifying parameters, etc.).

In [ ]:
```python
# Logistic regression class
class Logistic_regression:
    # Class constructor
    def __init__(self):
        self.w = None      # logistic regression weights
        self.saved_w = [] # Since this is a small problem, we can save the weights
                          #  at each iteration of gradient descent to build our
                          #  learning curves
        # returns nothing
        pass

    # Method for calculating the sigmoid function of w^T X for an input set of weights
    def sigmoid(self, X, w):
        # returns the value of the sigmoid
        pass

    # Cost function for an input set of weights
    def cost(self, X, y, w):
        # returns the average cross entropy cost
        pass

    # Update the weights in an iteration of gradient descent
    def gradient_descent(self, X, y, lr):
        # returns a scalar of the magnitude of the Euclidean norm
        #  of the change in the weights during one gradient descent step
        pass

    # Fit the logistic regression model to the data through gradient descent
    def fit(self, X, y, w_init, lr, delta_thresh=1e-6, max_iter=5000, verbose=False):
        # Note the verbose flag enables you to print out the weights at each iteration
        #  (optional - but may help with one of the questions)

        # returns nothing
        pass

    # Use the trained model to predict the confidence scores (prob of positive class in this case)
    def predict_proba(self, X):
        # returns the confidence score for the each sample
        pass

    # Use the trained model to make binary predictions
    def predict(self, X, thresh=0.5):
        # returns a binary prediction for each sample
        pass

    # Stores the learning curves from saved weights from gradient descent
    def learning_curve(self, X, y):
        # returns the value of the cost function from each step in gradient descent
        #  from the last model fitting process
        pass

    # Appends a column of ones as the first feature to account for the bias term
    def prepare_x(self, X):
        # returns the X with a new feature of all ones (a column that is the new column 0)
        pass
```

(h) Choose a learning rate and fit your model. Learning curves are a plot of metrics of model performance evaluated through the process of model training to provide insight about how model training is proceeding. Show the learning curves for the gradient descent process for learning rates of $\{10^{-0}, 10^{-2}, 10^{-4}\}$. For each learning rate plot the learning curves by plotting **both the training and test data average cost** as a function of each iteration of gradient descent. You should run the model fitting process until it completes (up to 5,000 iterations of gradient descent). Each of the 6 resulting curves (train and test average cost for each learning rate) should be plotted on the **same set of axes** to enable direct comparison. *Note: make sure you're using average cost per sample, not the total cost.*

- Try running this process for a really big learning rate for this problem: 10^2. Look at the weights that the fitting process generates over the first 50 iterations and how they change. Either print these first 50 iterations as console output or plot them. What happens? How does the output compare to that corresponding to a learning rate of 10^0 and why?
- What is the impact that the different values of learning have on the speed of the process and the results?
- Of the options explored, what learning rate do you prefer and why?
- Use your chosen learning rate for the remainder of this problem.

V. Evaluate your model performance through cross validation

(i) Test the performance of your trained classifier using K-folds cross validation resampling technique. The scikit-learn package StratifiedKFolds may be helpful.

- Train your logistic regression model and a K-Nearest Neighbor classification model with $k = 7$ nearest neighbors.
- Using the trained models, make four plots: two for logistic regression and two for KNN. For each model have one plot showing the training data used for fitting the model, and the other showing the test data. On each plot, include the decision boundary resulting from your trained classifier.
- Produce a Receiver Operating Characteristic curve (ROC curve) that represents the performance from cross validated performance evaluation for each classifier (your logistic regression model and the KNN model, with $k = 7$ nearest neighbors). For the cross validation, use $k = 10$ folds.
    - Plot these curves on the same set of axes to compare them
    - On the ROC curve plot, also include the chance diagonal for reference (this represents the performance of the worst possible classifier). This is represented as a line from $(0, 0)$ to $(1, 1)$.

- Calculate the Area Under the Curve for each model and include this measure in the legend of the ROC plot.

- Comment on the following:
  - What is the purpose of using cross validation for this problem?
  - How do the models compare in terms of performance (both ROC curves and decision boundaries) and which model (logistic regression or KNN) would you select to use on previously unseen data for this problem and why?

ANSWER

(a) Loading Data

In [ ]:
```
import pandas as pd
import numpy as np
import altair as alt
data = pd.read_csv('data\A3_Q1_data.csv')
```

In [ ]:
```
test = data.sample(frac = .3)
train = data.loc[~data.index.isin(test.index), :]
```

In [ ]:
```
%%capture --no-display
alt.Chart(
    train,
    title = "Data Separated by Class").mark_circle().encode(
        x='x1',
        y='x2',
        color=alt.Color(
            'y',
            scale=alt.Scale(
                domain = [0.0, 1.0],
                range = ['red', 'blue'])))
```

Out[ ]:

The Data is not linearly separable in this feature space. Logistic regression appears to be a reasonable choice for predicting classes, because the analysis is seeking binary classification and there appears to be an approximately linear decision boundary.

(b) Exploratory Data Analysis

In [ ]:
```
eda = pd.DataFrame(
    {'x1': [train.loc[:, 'x1'].min(),
        train.loc[:, 'x1'].max(),
        train.loc[:, 'x1'].max() - train.loc[:, 'x1'].min(),
        train.loc[:, 'x1'].mean(),
        train.loc[:, 'x1'].std()],
    'x2': [train.loc[:, 'x2'].min(),
        train.loc[:, 'x2'].max(),
        train.loc[:, 'x2'].max() - train.loc[:, 'x2'].min(),
        train.loc[:, 'x2'].mean(),
        train.loc[:, 'x2'].std()]},
    index = ['min', 'max', 'range', 'mean', 'std'])

eda
```

Out[ ]:

	x1	x2
min	-3.210005	-3.143907
max	3.867647	3.103541
range	7.077653	6.247447
mean	0.188612	-0.398786
std	1.417341	1.184144

While there are some minor differences in the training data exploratory statistics, the axes do not seem different enough to require pre-processing.

In [ ]:
```
print("The number of missing values in the training data is: ")
train.isna().sum()
```

```
The number of missing values in the training data is:
```

Out[ ]:
```
x1    0
x2    0
y     0
dtype: int64
```

There does not appear to be any missing values in the data.

In [ ]:
```
print(f"The range of values for class in the training data is {train.loc[:, 'y'].unique()}.")
```

```
The range of values for class in the training data is [0 1].
```

The y value data is appropriate for binary classification.

(c) Likelihood Function

The likelihood function associated with the entire dataset can be written as
$L(\mathbf{w}|\mathbf{y},\mathbf{x}) = \Pi_i^N P(y_i|\mathbf{x}_i) = \Pi_i^N P(y_i=1|\mathbf{x}_i)^{y_i} P(y_i=0|\mathbf{x}_i)^{1-y_i} = \Pi_i^N \sigma(\mathbf{w}^T\mathbf{x}_i)^{y_i}(1-\sigma(\mathbf{w}^T\mathbf{x}_i))^{1-y_i}.$

(d) Cost Function

$$C(\mathbf{w}) = -\frac{1}{N} log(L(\mathbf{w}|\mathbf{y},\mathbf{x})) =$$

$$-\frac{1}{N} log(\Pi_i^N \sigma(\mathbf{w}^T\mathbf{x}_i)^{y_i}(1-\sigma(\mathbf{w}^T\mathbf{x}_i))^{1-y_i}) =$$
$$-\frac{1}{N}\sum_i^N (y_i log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1-y_i) log(1-\sigma(\mathbf{w}^T\mathbf{x}_i)))$$

(e) Gradient of the Cost Function

$$\nabla_{\mathbf{w}}(C(\mathbf{w})) = \nabla_{\mathbf{w}}(-\frac{1}{N}\sum_i^N (y_i log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1-y_i) log(1-\sigma(\mathbf{w}^T\mathbf{x}_i)))) =$$

$$-\frac{1}{N}\sum_i^N (\nabla_{\mathbf{w}}(y_i log(\sigma(\mathbf{w}^T\mathbf{x}_i))) + \nabla_{\mathbf{w}}((1-y_i) log(1-\sigma(\mathbf{w}^T\mathbf{x}_i)))) =$$
$$-\frac{1}{N}\sum_i^N (\frac{y_i}{\sigma(\mathbf{w}^T\mathbf{x}_i)}\nabla_{\mathbf{w}}(\sigma(\mathbf{w}^T\mathbf{x}_i)) + \frac{1-y_i}{1-\sigma(\mathbf{w}^T\mathbf{x}_i)}\nabla_{\mathbf{w}}(1-\sigma(\mathbf{w}^T\mathbf{x}_i))) =$$
$$-\frac{1}{N}\sum_i^N (\frac{y_i}{\sigma(\mathbf{w}^T\mathbf{x}_i)}\sigma(\mathbf{w}^T\mathbf{x}_i)(1-\sigma(\mathbf{w}^T\mathbf{x}_i))\nabla_{\mathbf{w}}(\mathbf{w}^T\mathbf{x}_i) + \frac{1-y_i}{1-\sigma(\mathbf{w}^T\mathbf{x}_i)}\sigma(\mathbf{w}^T\mathbf{x}_i)(\sigma(\mathbf{w}^T\mathbf{x}_i)-1)\nabla_{\mathbf{w}}(\mathbf{w}^T\mathbf{x}_i)) =$$
$$-\frac{1}{N}\sum_i^N (\frac{y_i}{\sigma(\mathbf{w}^T\mathbf{x}_i)}\sigma(\mathbf{w}^T\mathbf{x}_i)(1-\sigma(\mathbf{w}^T\mathbf{x}_i))\mathbf{x}_i + \frac{y_i-1}{1-\sigma(\mathbf{w}^T\mathbf{x}_i)}\sigma(\mathbf{w}^T\mathbf{x}_i)(1-\sigma(\mathbf{w}^T\mathbf{x}_i))\mathbf{x}_i) =$$
$$-\frac{1}{N}\sum_i^N (y_i - \sigma(\mathbf{w}^T\mathbf{x}_i))\mathbf{x}_i$$

(f) Gradient Descent Function

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \eta\nabla_{\mathbf{w}}(C(\mathbf{w})) = \mathbf{w}_n + \eta\frac{1}{N}\sum_i^N (2y_i - 1)(1-\sigma(\mathbf{w}^T\mathbf{x}_i))\mathbf{x}_i$$

(g) Defining Logistic Regression Model

In [ ]:
```
# Logistic regression class
class Logistic_regression:
    """This is a class for a logistic regression model. It is initialized with no parameters."""
    # Class constructor
    def __init__(self):
        self.w = None     # logistic regression weights
        self.saved_w = [] # Since this is a small problem, we can save the weights
                          #  at each iteration of gradient descent to build our
                          #  learning curves
        # returns nothing
        pass

    # Method for calculating the sigmoid function of w^T X for an input set of weights
    def sigmoid(self, X, w):
        """Method for calculating the sigmoid function of w^T X for an input set of weights."""
        return 1 / (1 + np.exp(-np.matmul(X, w.T)))

    # Cost function for an input set of weights
    def cost(self, X, y, w):
        """Method for calculating the cost function for an input set of weights."""
        N = X.shape[0]
        return -1 / N * np.sum(y * np.log(self.sigmoid(X, w)) + (1 - y) * np.log(1 - self.sigmoid(X, w)))

    # Update the weights in an iteration of gradient descent
    def gradient_descent(self, X, y, lr):
        """Method for updating the weights in an iteration of gradient descent."""
        N = X.shape[0]
        return self.w + lr / N * np.sum(np.multiply(y - self.sigmoid(X, self.w),  X), axis = 0)

    # Fit the logistic regression model to the data through gradient descent
    def fit(self, X, y, lr, w_init = None, delta_thresh=1e-6, max_iter=5000, verbose=False):
        """Method for fitting the logistic regression model to the data through gradient
        descent."""
        X = self.prepare_x(X)
        if w_init is None:
            self.w = np.random.rand(y.shape[1] if len(y.shape) > 1 else 1, X.shape[1])
        else:
            self.w = w_init

        self.saved_w.append(self.w)

        for i in range(max_iter):
            self.w = self.gradient_descent(
                X,
                np.reshape(
```

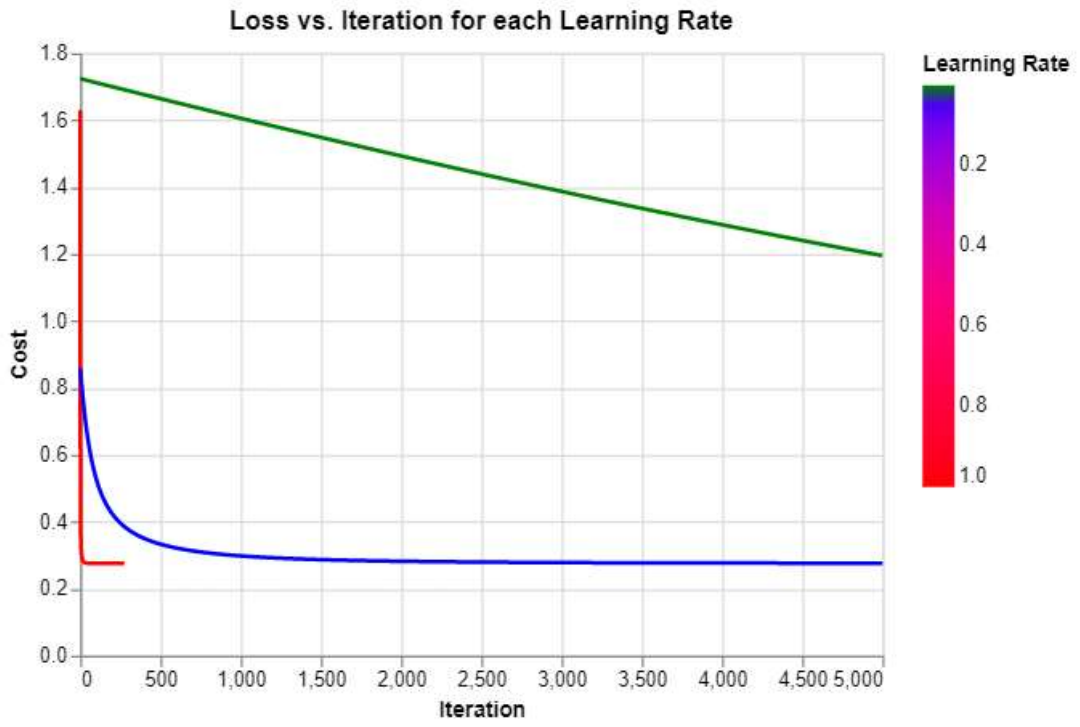
```
                y.to_numpy(),
                (-1, 1)),
            lr)

        if np.linalg.norm(self.w - self.saved_w[-1], 2) < delta_thresh:
            break

        self.saved_w.append(self.w)

# Use the trained model to predict the confidence scores (prob of positive class in this case)
def predict_proba(self, X):
    """Method for using the trained model to predict the confidence scores
    (prob of positive class in this case)."""
    return self.sigmoid(
        self.prepare_x(X),
        self.w).reshape(1, -1)[0]

# Use the trained model to make binary predictions
def predict(self, X, thresh=0.5):
    """Method for using the trained model to make binary predictions."""
    return np.where(self.predict_proba(X) > thresh, 1, 0)

# Stores the learning curves from saved weights from gradient descent
def learning_curve(self, X, y):
    """Method for storing the learning curves from saved weights from gradient descent."""
    return np.array(
        [self.cost(
            self.prepare_x(X),
            np.reshape(y.to_numpy(), (-1, 1)),
            each_w) for each_w in self.saved_w])

# Appends a column of ones as the first feature to account for the bias term
def prepare_x(self, X):
    """Method for appending a column of ones as the first feature to account for the
    bias term."""
    return np.hstack([np.ones((X.shape[0], 1)), X])

def __sklearn_is_fitted__(self):
    """Method for checking if the model has been fitted."""
    if self.w is not None:
        return True
    else:
        return False
```

(h) Implementing Logistic Regression Model

In [ ]:

```
%%capture --no-display
lrates = [1, .01, .0001]
plots = None
for lr in lrates:
    model = Logistic_regression()

    model.fit(
        train.loc[:, ['x1', 'x2']],
        train.loc[:, 'y'],
        lr)

    if plots is not None:

        plots = pd.concat(
            [plots,
            pd.DataFrame(
                {lr: model.learning_curve(
                    train.loc[:, ['x1', 'x2']],
                    train.loc[:, 'y'])})],
                    axis = 1)

        model.learning_curve(
            test.loc[:, ['x1', 'x2']],
            test.loc[:, 'y'])

    else:

        plots = pd.DataFrame(
            {lr: model.learning_curve(
                train.loc[:, ['x1', 'x2']],
                train.loc[:, 'y'])})

        model.learning_curve(
            test.loc[:, ['x1', 'x2']],
            test.loc[:, 'y'])

alt.data_transformers.disable_max_rows()

alt.Chart(plots.reset_index(names = ['Iteration']).melt(
    id_vars = 'Iteration',
    var_name = 'Learning Rate',
    value_name = 'Cost'),
        title = "Loss vs. Iteration for each Learning Rate").mark_line().encode(
            x = 'Iteration',
            y = 'Cost',
            color = alt.Color('Learning Rate', scale = alt.Scale(
                domain = lrates,
                range = ['red', 'blue', 'green'])))
```

Out[ ]:



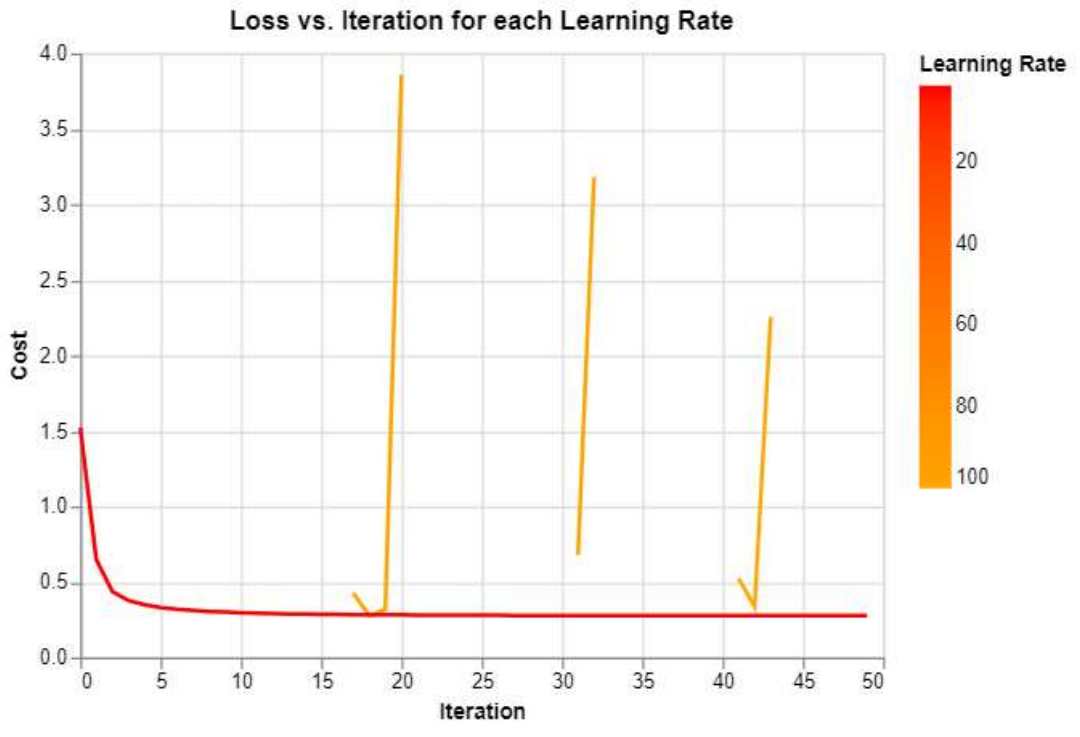
The higher learning rate (when only comparing 1, .01, and .0001) allows the gradient descent to converge much faster. Over the course of 5,000 iterations, only the learning rate of 1 converges before the 5,000 iterations (converging after only ~400 iterations). It would be preferable to use the higher learning rate, because in a more complicated gradient descent problem where computation can be very time consuming, the higher learning rate will allow for convergence in a substantially more reasonable time.

In [ ]:

```
lrates = [100, 1]
plots = []
for lr in lrates:
    model = Logistic_regression()

    model.fit(
        train.loc[:, ['x1', 'x2']],
        train.loc[:, 'y'],
        lr,
        max_iter = 50 if lr == 100 else 5000)

    if not isinstance(plots, list):

        plots = pd.concat(
            [plots,
            pd.DataFrame(
                {lr: model.learning_curve(
                    train.loc[:, ['x1', 'x2']],
                    train.loc[:, 'y'])})],
                    axis = 1)

    else:

        plots = pd.DataFrame(
            {lr: model.learning_curve(train.loc[:, ['x1', 'x2']],
            train.loc[:, 'y'])})

plots2 = plots.reset_index(names = ['Iteration']).melt(
    id_vars = 'Iteration',
    var_name = 'Learning Rate',
    value_name = 'Cost')

alt.Chart(plots2.loc[plots2.loc[:, 'Iteration'] < 50, :],
        title = "Loss vs. Iteration for each Learning Rate").mark_line().encode(
            x = 'Iteration',
            y = 'Cost',
            color = alt.Color('Learning Rate', scale = alt.Scale(
                domain = lrates,
                range = ['orange', 'red'])))
```

```
C:\Users\nicho\AppData\Local\Temp\ipykernel_29656\4138717402.py:22: RuntimeWarning: divide by zero encountered in log
  return -1 / N * np.sum(y * np.log(self.sigmoid(X, w)) + (1 - y) * np.log(1 - self.sigmoid(X, w)))
C:\Users\nicho\AppData\Local\Temp\ipykernel_29656\4138717402.py:22: RuntimeWarning: invalid value encountered in multiply
  return -1 / N * np.sum(y * np.log(self.sigmoid(X, w)) + (1 - y) * np.log(1 - self.sigmoid(X, w)))
c:\Users\nicho\miniconda3\lib\site-packages\altair\utils\core.py:317: FutureWarning: iteritems is deprecated and will be removed in a future version. Use .items instead.
  for col_name, dtype in df.dtypes.iteritems():
```

Out[ ]:



With a learning rate as high as 100, the change in the parameters causes oscillating loss, jumping over the minimum when it gets close, and exploding. Using the correct learning rate has a dramatic impact on how quickly the training data is able to converge, and if it is able to converge at all. The most appropriate learning rate for this problem is 1.

In [ ]:
```
plots.iloc[:50]
```

Out[ ]:

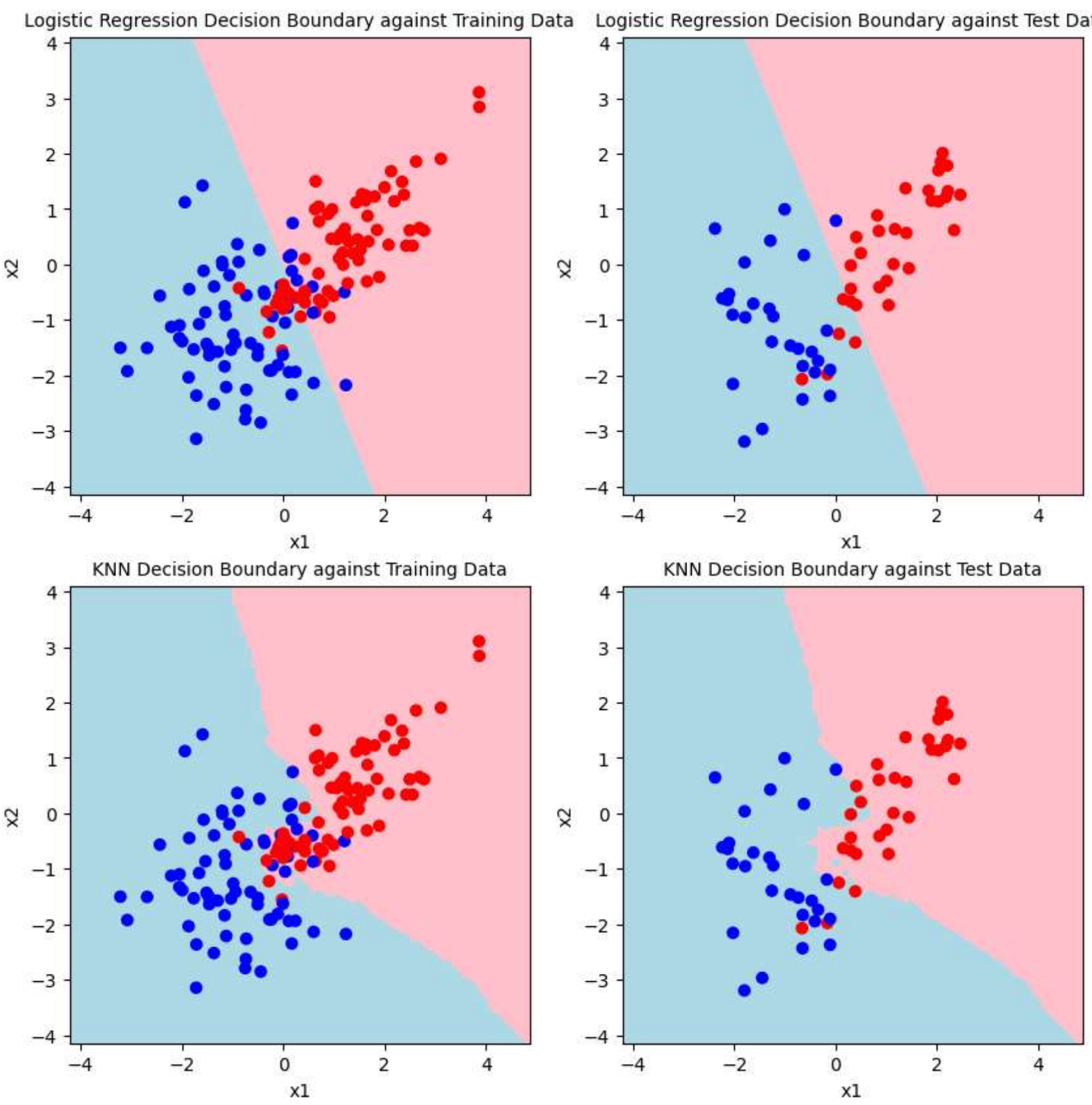
	100	1
0	1.072138	1.521594
1	NaN	0.647003
2	NaN	0.436785
3	NaN	0.377732
4	NaN	0.348568
5	NaN	0.331036
6	NaN	0.319358
7	NaN	0.311067
8	NaN	0.304914
9	NaN	0.300197
10	NaN	0.296490
11	NaN	0.293519
12	NaN	0.291098
13	NaN	0.289099
14	NaN	0.287431
15	NaN	0.286024
16	NaN	0.284828
17	0.426800	0.283804
18	0.276988	0.282922
19	0.318052	0.282158
20	3.858952	0.281492
21	NaN	0.280910
22	NaN	0.280398
23	NaN	0.279948
24	NaN	0.279549
25	NaN	0.279195
26	NaN	0.278881
27	NaN	0.278600
28	NaN	0.278350
29	NaN	0.278125
30	NaN	0.277924
31	0.678121	0.277743
32	3.181697	0.277580
33	NaN	0.277433
34	NaN	0.277300
35	NaN	0.277180
36	NaN	0.277071
37	NaN	0.276972
38	NaN	0.276882
39	NaN	0.276801
40	NaN	0.276727
41	0.523774	0.276659
42	0.336186	0.276598
43	2.253452	0.276542
44	NaN	0.276491
45	NaN	0.276444
46	NaN	0.276401
47	NaN	0.276362
48	NaN	0.276326
49	NaN	0.276293

(i) Model Performance Evaluation

In [ ]:
```
%%capture --no-display

from sklearn.model_selection import StratifiedKFold
from sklearn.inspection import DecisionBoundaryDisplay
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
```

```
# Fitting logistic regression model
logmodel = Logistic_regression()
logmodel.fit(train.loc[:, ['x1', 'x2']], train.loc[:, 'y'], 1)

# Fitting KNN model
k = 7
knnmodel = KNeighborsClassifier(n_neighbors = k)

knnmodel.fit(
    train.loc[:, ['x1', 'x2']].to_numpy(),
    train.loc[:, 'y'].to_numpy())

# Setting up plots
fig, axs = plt.subplots(2, 2, figsize = (10, 10))

# Plotting logistic regression decision boundaries
DecisionBoundaryDisplay.from_estimator(
    logmodel,
    train.loc[:, ['x1', 'x2']],
    response_method = 'predict_proba',
    cmap = ListedColormap(['pink', 'lightblue']),
    ax = axs[0, 0])

DecisionBoundaryDisplay.from_estimator(
    logmodel,
    train.loc[:, ['x1', 'x2']],
    response_method = 'predict_proba',
    cmap = ListedColormap(['pink', 'lightblue']),
    ax = axs[0, 1])

axs[0, 0].scatter(
    train.loc[:, 'x1'],
    train.loc[:, 'x2'],
    c = train.loc[:, 'y'],
    cmap = ListedColormap(['red', 'blue']))

axs[0, 0].set_title(
    'Logistic Regression Decision Boundary against Training Data',
    fontsize = 10)

axs[0, 1].scatter(
    test.loc[:, 'x1'],
    test.loc[:, 'x2'],
    c = test.loc[:, 'y'],
    cmap = ListedColormap(['red', 'blue']))

axs[0, 1].set_title(
    'Logistic Regression Decision Boundary against Test Data',
    fontsize = 10)

# Plotting KNN decision boundaries
DecisionBoundaryDisplay.from_estimator(
    knnmodel,
    train.loc[:, ['x1', 'x2']],
    response_method = 'predict_proba',
    cmap = ListedColormap(['pink', 'lightblue']),
    ax = axs[1, 0])

DecisionBoundaryDisplay.from_estimator(
    knnmodel,
    train.loc[:, ['x1', 'x2']],
    response_method = 'predict_proba',
    cmap = ListedColormap(['pink', 'lightblue']),
    ax = axs[1, 1])

axs[1, 0].scatter(
    train.loc[:, 'x1'],
    train.loc[:, 'x2'],
    c = train.loc[:, 'y'],
    cmap = ListedColormap(['red', 'blue']))

axs[1, 0].set_title('KNN Decision Boundary against Training Data', fontsize = 10)

axs[1, 1].scatter(
    test.loc[:, 'x1'],
    test.loc[:, 'x2'],
    c = test.loc[:, 'y'],
    cmap = ListedColormap(['red', 'blue']))

axs[1, 1].set_title('KNN Decision Boundary against Test Data', fontsize = 10)
```

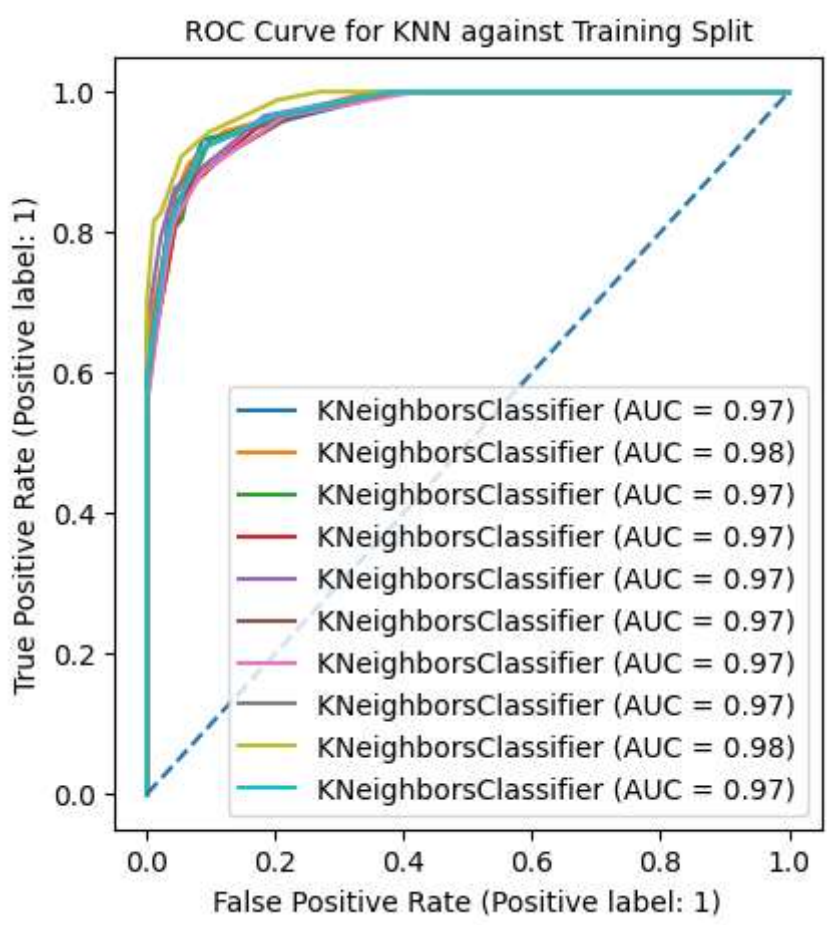
Out[ ]: Text(0.5, 1.0, 'KNN Decision Boundary against Test Data')

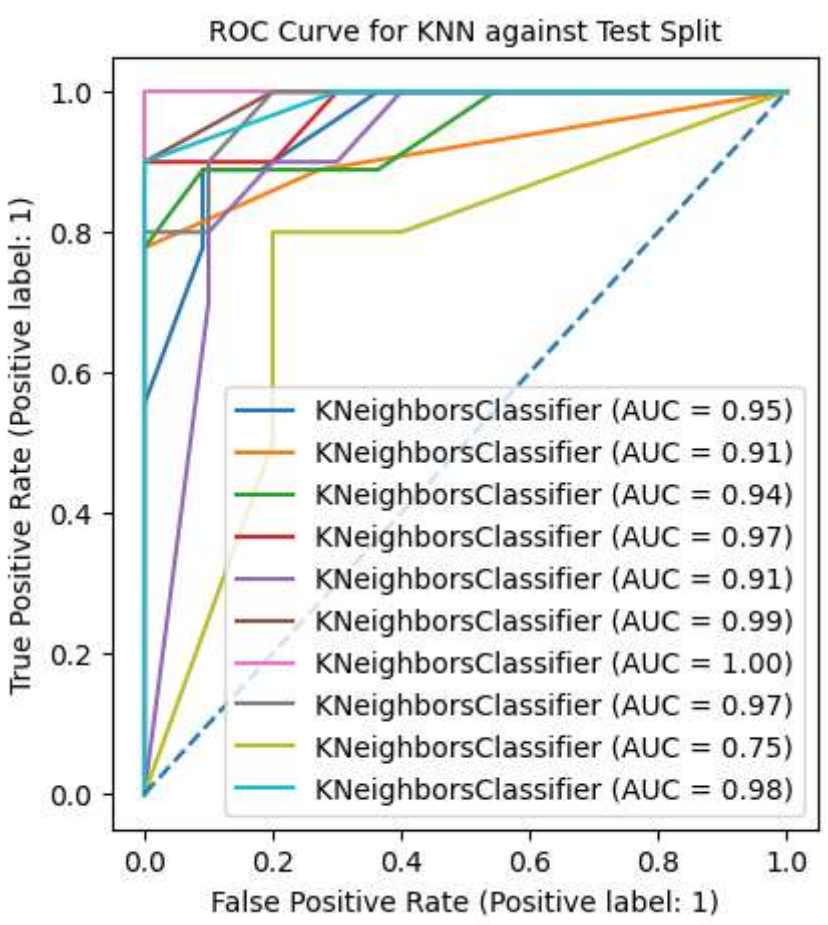


In [ ]:
```
from sklearn.metrics import RocCurveDisplay
```

In [ ]:
```
%%capture --no-display
fig, axs = plt.subplots(1, 2, figsize = (10, 5))

skf = StratifiedKFold(n_splits = 10)

for train_index, test_index in skf.split(data.loc[:, ['x1', 'x2']], data.loc[:, 'y']):
    knnmodel = KNeighborsClassifier(n_neighbors = k)

    knnmodel.fit(
        data.loc[train_index, ['x1', 'x2']].to_numpy(),
        data.loc[train_index, 'y'].to_numpy())

    RocCurveDisplay.from_estimator(
        knnmodel,
        data.loc[train_index, ['x1', 'x2']],
        data.loc[train_index, 'y'],
        response_method='predict_proba',
        ax = axs[0])

    RocCurveDisplay.from_estimator(
        knnmodel,
        data.loc[test_index, ['x1', 'x2']],
        data.loc[test_index, 'y'],
        response_method='predict_proba',
        ax = axs[1])

# Plotting the no skill line
axs[0].plot([0, 1], [0, 1], linestyle='--', label='No Skill')
axs[1].plot([0, 1], [0, 1], linestyle='--', label='No Skill')

# Adding Titles
axs[0].set_title('ROC Curve for KNN against Training Split', fontsize = 10)
axs[1].set_title('ROC Curve for KNN against Test Split', fontsize = 10)
```

Out[ ]:
```
Text(0.5, 1.0, 'ROC Curve for KNN against Test Split')
```

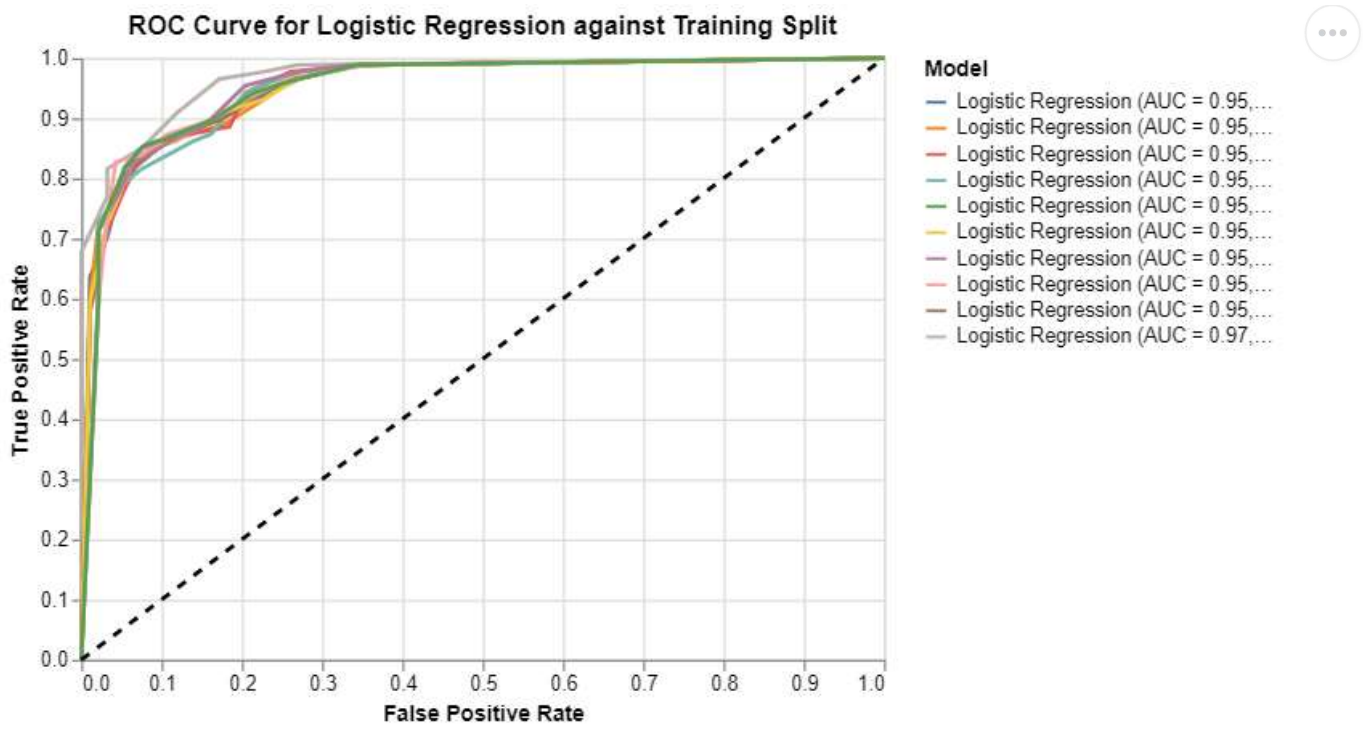


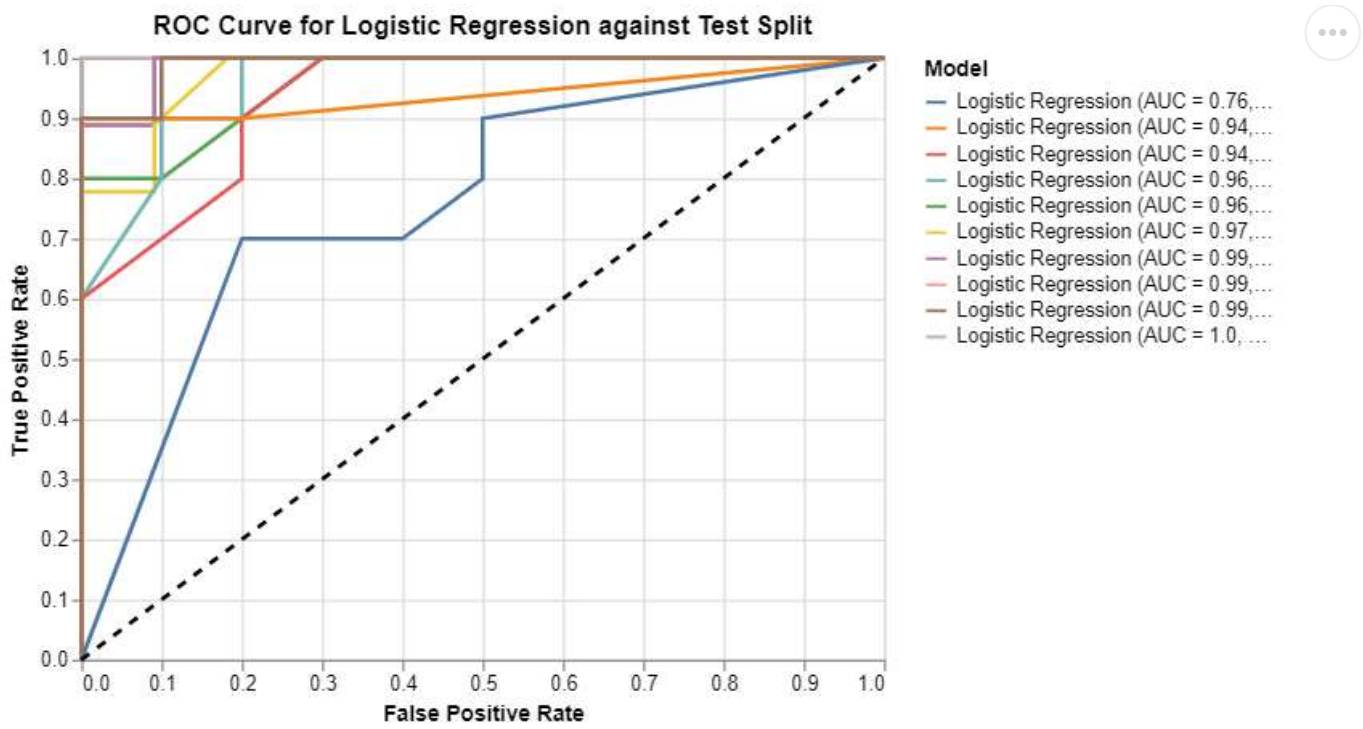


In [ ]:

```
%%capture --no-display

# Creating a dictionary to store the ROC curves
rocdict_train = {}
rocdict_test = {}

# Putting the ROC curves into the dictionary
for fold, (train_index, test_index) in enumerate(
    skf.split(
        data.loc[:, ['x1', 'x2']],
        data.loc[:, 'y'])):
    logmodel = Logistic_regression()
    logmodel.fit(
        data.loc[train_index, ['x1', 'x2']],
        data.loc[train_index, 'y'], 1)

    for num, eachSplit in enumerate((train_index, test_index)):
        positives = logmodel.predict_proba(
            data.loc[eachSplit, ['x1', 'x2']].loc[data.loc[eachSplit, 'y'] == 1, :])
        negatives = logmodel.predict_proba(
            data.loc[eachSplit, ['x1', 'x2']].loc[data.loc[eachSplit, 'y'] == 0, :])
        templist = {'True Positive Rate': [], 'False Positive Rate': []}
        for i in range(11):
            templist['True Positive Rate'].append(
                (positives > i / 10.0).sum() / positives.shape[0])
            templist['False Positive Rate'].append(
                (negatives > i / 10.0).sum() / negatives.shape[0])

        dftemplist = pd.DataFrame(templist)

        auc = round(np.dot(
            ((dftemplist.loc[:, 'True Positive Rate'] + dftemplist.loc[:,
                'True Positive Rate'].shift(1)) / 2).dropna(),
            (-dftemplist.loc[:, 'False Positive Rate'] + dftemplist.loc[:,
                'False Positive Rate'].shift(1)).dropna()), 2)
        if num == 0:
            rocdict_train[
                (f"Logistic Regression (AUC = {auc}, K-fold = {fold})", 'True Positive Rate')] = templist['True Positive Rate']
            rocdict_train[
                (f"Logistic Regression (AUC = {auc}, K-fold = {fold})", 'False Positive Rate')] = templist['False Positive Rate']
        else:
            rocdict_test[
                (f"Logistic Regression (AUC = {auc}, K-fold = {fold})", 'True Positive Rate')] = templist['True Positive Rate']
            rocdict_test[
                (f"Logistic Regression (AUC = {auc}, K-fold = {fold})", 'False Positive Rate')] = templist['False Positive Rate']

# Converting the dictionary into a dataframe
rocdf_train = pd.DataFrame(rocdict_train).stack(level = 0).reset_index(
    names = ['drop', 'Model'])
rocdf_test = pd.DataFrame(rocdict_test).stack(level = 0).reset_index(
    names = ['drop', 'Model'])

# Plotting the ROC curves
trainChart = alt.Chart(
    rocdf_train.sort_values("True Positive Rate"),
    title = "ROC Curve for Logistic Regression against Training Split").mark_line().encode(
        x = 'False Positive Rate',
        y = 'True Positive Rate',
        color = 'Model')

testChart = alt.Chart(
    rocdf_test.sort_values("True Positive Rate"),
    title = "ROC Curve for Logistic Regression against Test Split").mark_line().encode(
        x = 'False Positive Rate',
        y = 'True Positive Rate',
        color = 'Model')
```

```
# Displaying the ROC curves
display(
    trainChart + alt.Chart(
        pd.DataFrame(
            {'False Positive Rate': [0, 1],
            'True Positive Rate': [0, 1]})).mark_line(
                color = 'black',
                strokeDash = [5, 5]).encode(
                    x = 'False Positive Rate',
                    y = 'True Positive Rate'))

display(
    testChart + alt.Chart(pd.DataFrame(
        {'False Positive Rate': [0, 1],
        'True Positive Rate': [0, 1]})).mark_line(
            color = 'black',
            strokeDash = [5, 5]).encode(
                x = 'False Positive Rate',
                y = 'True Positive Rate'))
```





Cross Validation allows us to compare how these two models generalize. While the KNN model performs slightly better on the training data than the logistic regression model, according to its AUC values, the two models seem to perform very similarly on the test data, with primarily high AUC values, but substantial variability among the parameters. The two models have very similar range in the AUC parameter on the test data. Either method seems equally sufficient to be used on unseen data; therefore, I would prefer to use the logistic regression model as I compared it to new data, because it is less computationally expensive to predict, and much easier to interpret.

2

Digits classification

[30 points]

An exploration of regularization, imbalanced classes, ROC and PR curves

The goal of this exercise is to apply your supervised learning skills on a very different dataset: in this case, image data; MNIST: a collection of images of handwritten digits. Your goal is to train a classifier that is able to distinguish the number "3" from all possible numbers and to do so as accurately as possible. You will first explore your data (this should always be your starting point to gain domain knowledge about the problem.). Since the feature space in this problem is 784-dimensional, overfitting is possible. To avoid overfitting you will investigate the impact of regularization on generalization performance (test accuracy) and compare regularized and unregularized logistic regression model test error against other classification techniques such as linear discriminant analysis and random forests and draw conclusions about the best-performing model.

Start by loading your dataset from the MNIST dataset of handwritten digits, using the code provided below. MNIST has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image.

Your goal is to classify whether or not an example digit is a 3. Your binary classifier should predict $y = 1$ if the digit is a 3, and $y = 0$ otherwise. Create your dataset by transforming your labels into a binary format (3's are class 1, and all other digits are class 0).

(a) Plot 10 examples of each class (i.e. class $y = 0$, which are not 3's and class $y = 1$ which are 3's), from the training dataset.

- Note that the data are composed of samples of length 784. These represent 28 x 28 images, but have been reshaped for storage convenience. To plot digit examples, you'll need to reshape the data to be 28 x 28 (which can be done with numpy `reshape`).

(b) How many examples are present in each class? Show a plot of samples by class (bar plot). What fraction of samples are positive? What issues might this cause?

(c) Identify the value of the regularization parameter that optimizes model performance on out-of-sample data. Using a logistic regression classifier, apply lasso regularization and retrain the model and evaluate its performance on the test set over a range of values on the regularization coefficient. You can implement this using the LogisticRegression module and activating the 'l1' penalty; the parameter C is the inverse of the regularization strength. Vary the value of C logarithmically from 10^{-4} to 10^{4} (and make your x-axes logarithmic in scale) and evaluate it at least 20 different values of C. As you vary the regularization coefficient, Plot the following four quantities (this should result in 4 separate plots)...

- The number of model parameters that are estimated to be nonzero (in the logistic regression model, one attribute is `coef_`, which gives you access to the model parameters for a trained model)
- The cross entropy loss (which can be evaluated with the Scikit Learn `log_loss` function)
- Area under the ROC curve (AUC)
- The F_1-score (assuming a threshold of 0.5 on the predicted confidence scores, that is, scores above 0.5 are predicted as Class 1, otherwise Class 0). Scikit Learn also has a `f1_score` function which may be useful.

-Which value of C seems best for this problem? Please select the closest power of 10. You will use this in the next part of this exercise.

(d) Train and test a (1) logistic regression classifier with minimal regularization (using the Scikit Learn package, set penalty='l1', C=1e100 to approximate this), (2) a logistic regression classifier with the best value of the regularization parameter from the last section, (3) a Linear Discriminant Analysis (LDA) Classifier, and (4) a Random Forest (RF) classifier (using default parameters for the LDA and RF classifiers).

- Compare your classifiers' performance using ROC and Precision Recall (PR) curves. For the ROC curves, all your curves should be plotted on the same set of axes so that you can directly compare them. Please do the same wih the PR curves.
- Plot the line that represents randomly guessing the class (50% of the time a "3", 50% not a "3"). You SHOULD NOT actually create random guesses. Instead, you should think through the theory behind how ROC and PR curves work and plot the appropriate lines. It's a good practice to include these in ROC and PR curve plots as a reference point.
- For PR curves, an excellent resource on how to correctly plot them can be found here (ignore the section on "non-linear interpolation between two points"). This describes how a random classifier is represented in PR curves and demonstrates that it should provide a lower bound on performance.
- When training your logistic regression model, it's recommended that you use solver="liblinear"; otherwise, your results may not converge.
- Describe the performance of the classifiers you compared. Did the regularization of the logistic regression model make much difference here? Which classifier you would select for application to unseen data.

In [ ]:
```
# Load the MNIST Data
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Set this to True to download the data for the first time and False after the first time
#   so that you just load the data locally instead
download_data = True

if download_data:
    # Load data from https://www.openml.org/d/554
    X, y = fetch_openml('mnist_784', return_X_y=True, as_frame=False)

    # Adjust the labels to be '1' if y==3, and '0' otherwise
    y[y!='3'] = 0
    y[y=='3'] = 1
    y = y.astype('int')

    # Divide the data into a training and test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/7, random_state=88)

    file = open('tmpdata', 'wb')
    pickle.dump((X_train, X_test, y_train, y_test), file)
    file.close()
else:
    file = open('tmpdata', 'rb')
    X_train, X_test, y_train, y_test = pickle.load(file)
    file.close()
```

ANSWER

Loading Data

In [ ]:
```
# Load the MNIST Data
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Set this to True to download the data for the first time and False after the first time
```

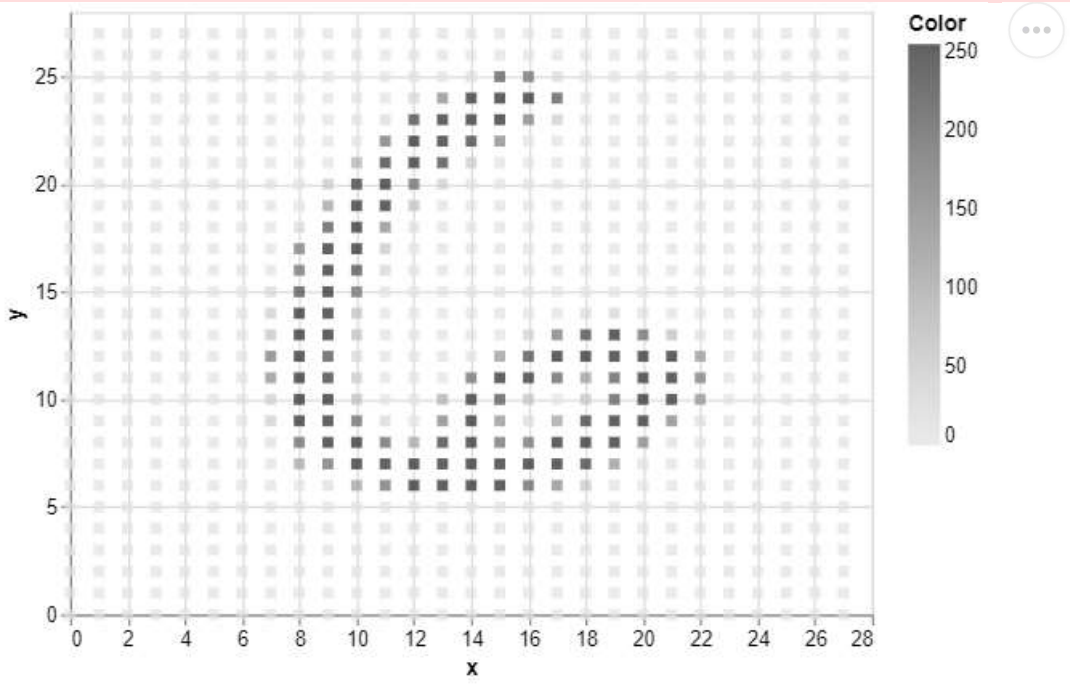
```
#   so that you just load the data locally instead
download_data = False

if download_data:
    # Load data from https://www.openml.org/d/554
    X, y = fetch_openml('mnist_784', return_X_y=True, as_frame=False)

    # Adjust the labels to be '1' if y==3, and '0' otherwise
    y[y!='3'] = 0
    y[y=='3'] = 1
    y = y.astype('int')

    # Divide the data into a training and test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/7, random_state=88)

    file = open('tmpdata', 'wb')
    pickle.dump((X_train, X_test, y_train, y_test), file)
    file.close()
else:
    file = open('tmpdata', 'rb')
    X_train, X_test, y_train, y_test = pickle.load(file)
    file.close()
```

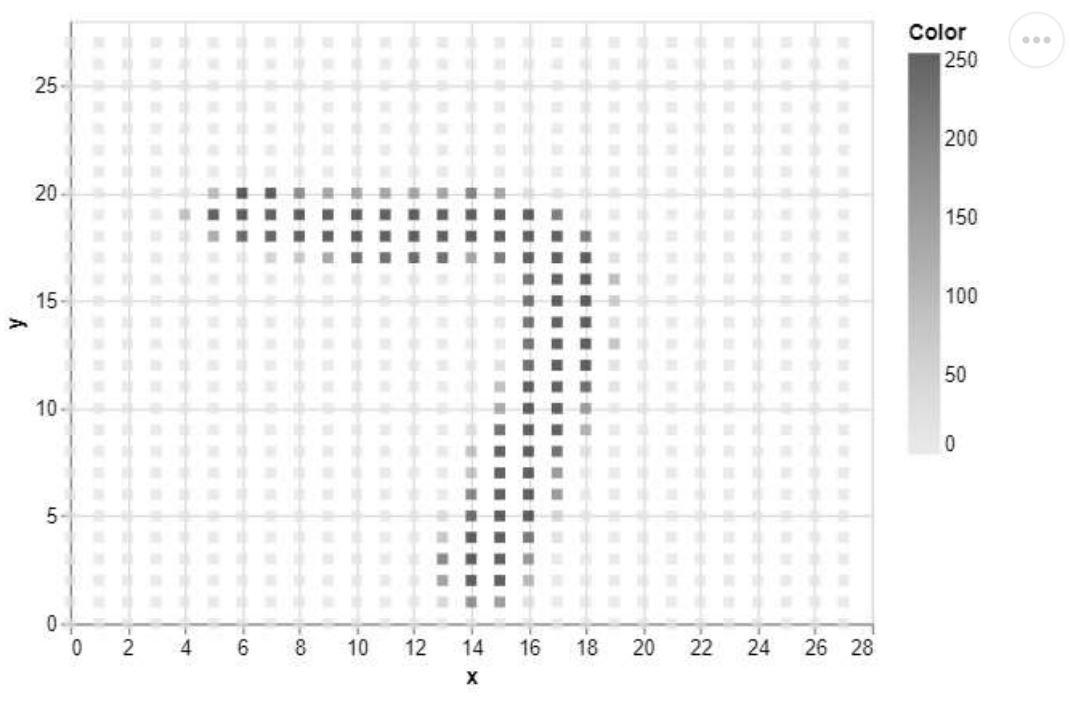
(a) Plotting the Data

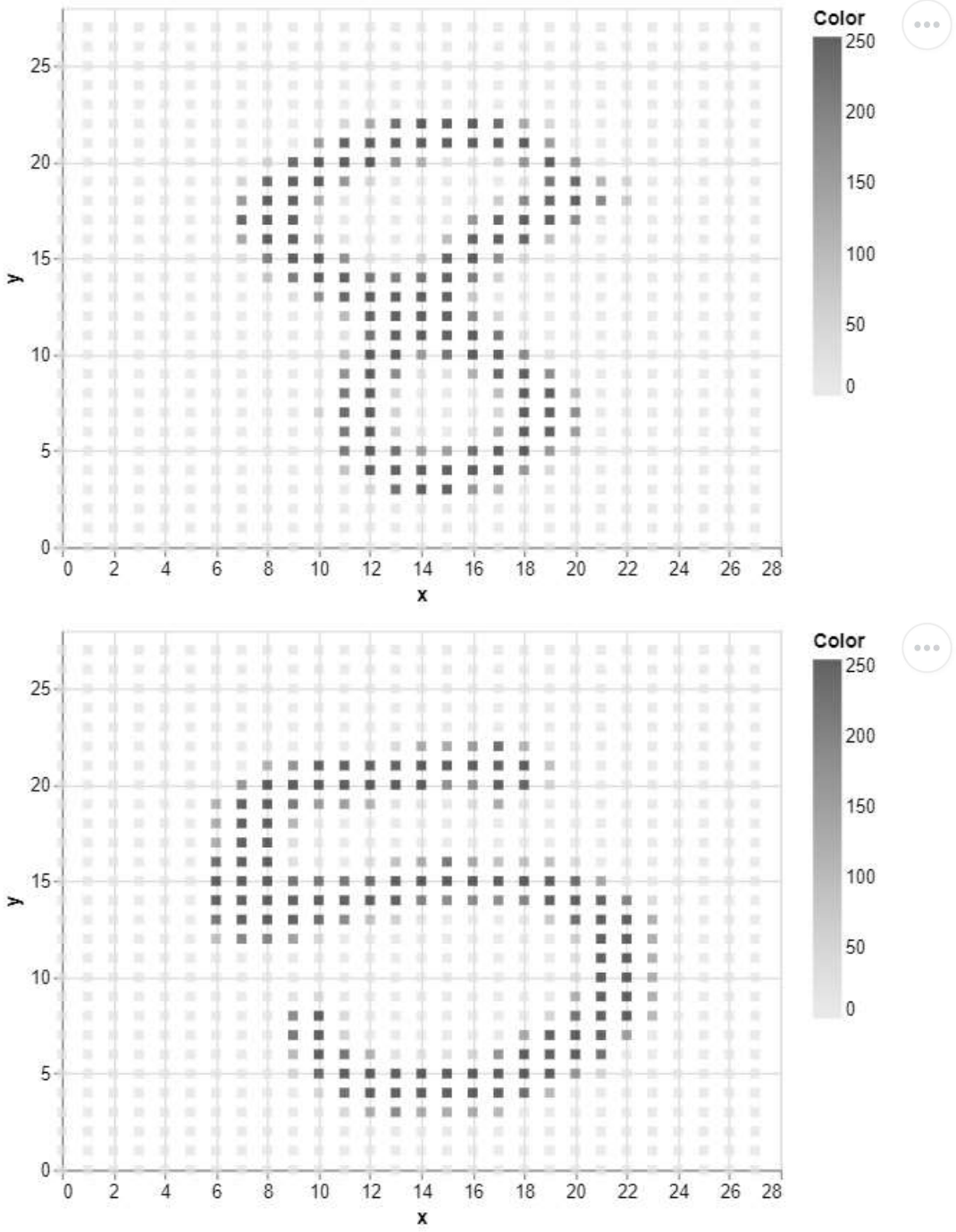
In [ ]:
```
images = []
for eachClass in range(2):
    reshaped_images = np.flip(
        X_train[y_train == eachClass][np.random.randint(
            0,
            X_train[y_train == eachClass].shape[0],
            10)].reshape(10, 28, 28), axis = 1)
    for eachImage in reshaped_images:
        images.append(
            alt.Chart(
                pd.DataFrame(
                    eachImage).stack().reset_index().rename(
                        columns = {
                            'level_0': 'y',
                            'level_1': 'x',
                            0: 'Color'})).mark_square().encode(
                                x = 'x',
                                y = 'y',
                                color = alt.Color(
                                    'Color:Q',
                                    scale = alt.Scale(scheme = 'greys'))))
```

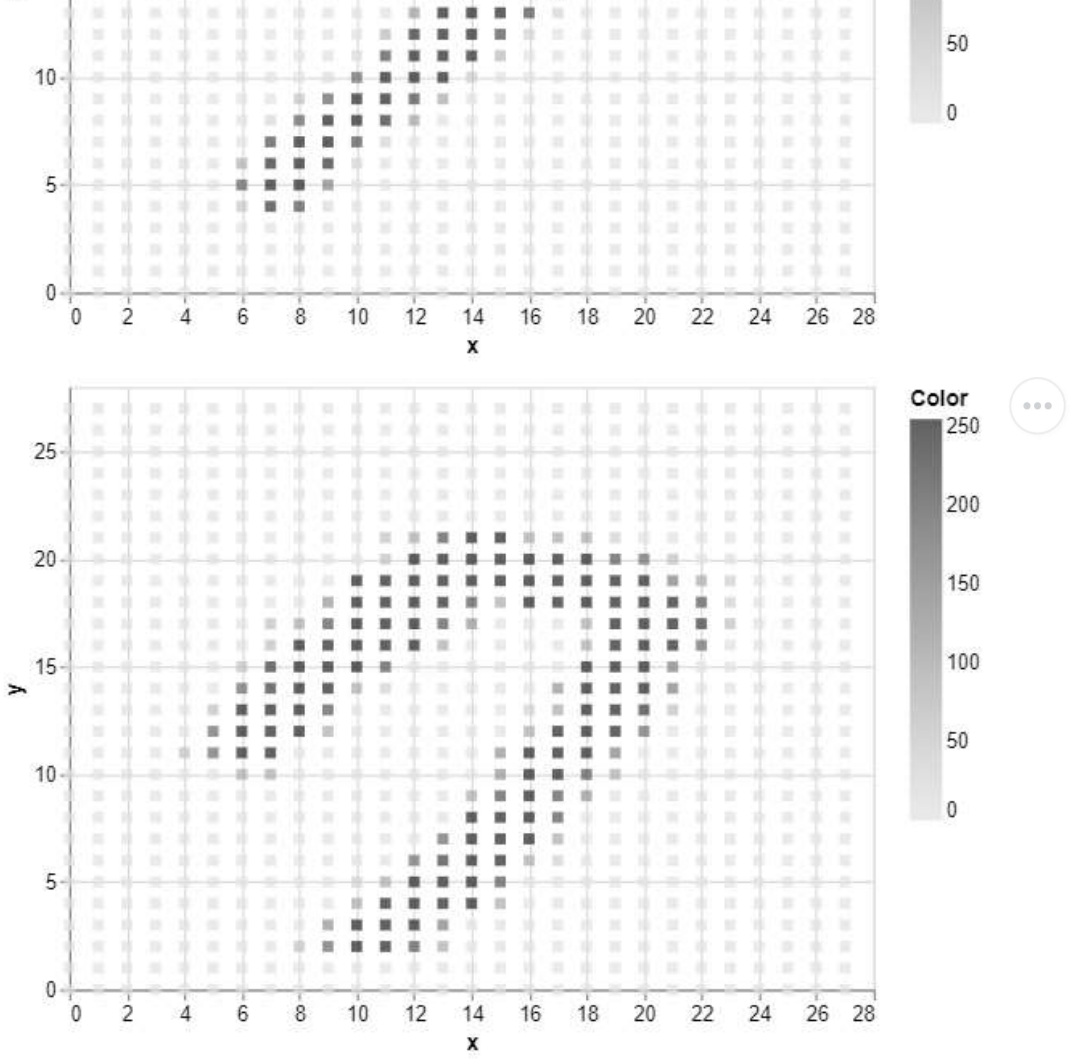
In [ ]:
```
for eachImage in images:
    display(eachImage)
```

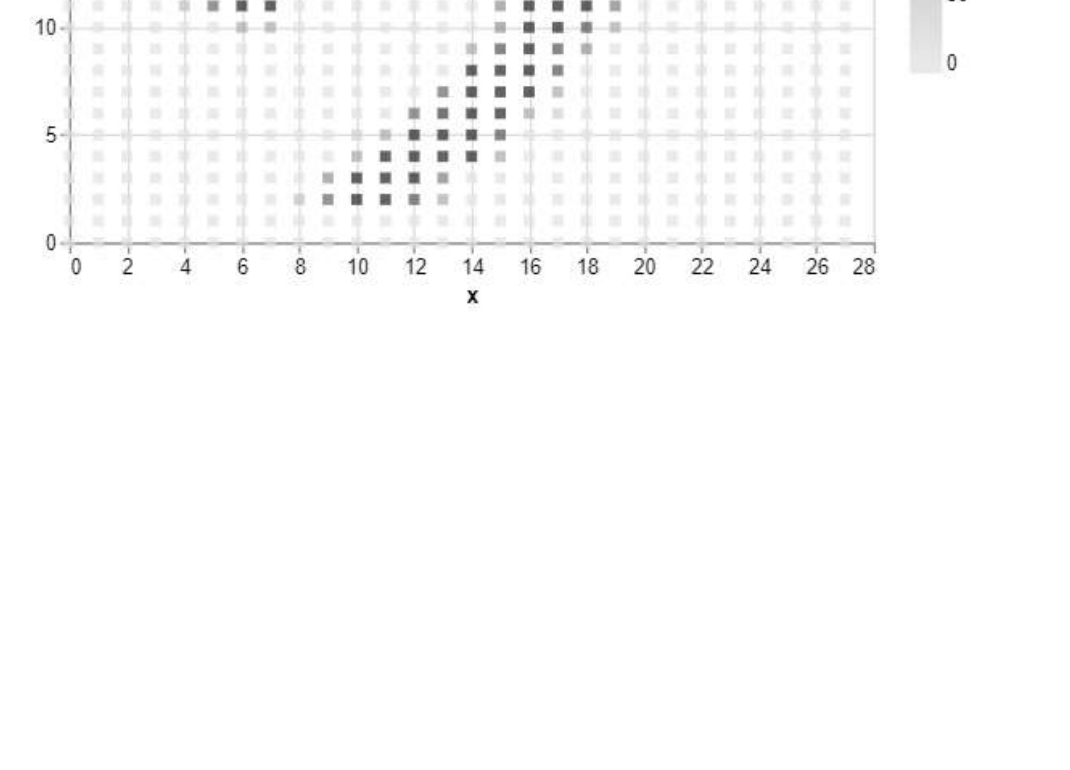
```
c:\Users\nicho\miniconda3\lib\site-packages\altair\utils\core.py:317: FutureWarning: iteritems is deprecated and will be removed in a future version. Use .items instead.
  for col_name, dtype in df.dtypes.iteritems():
```

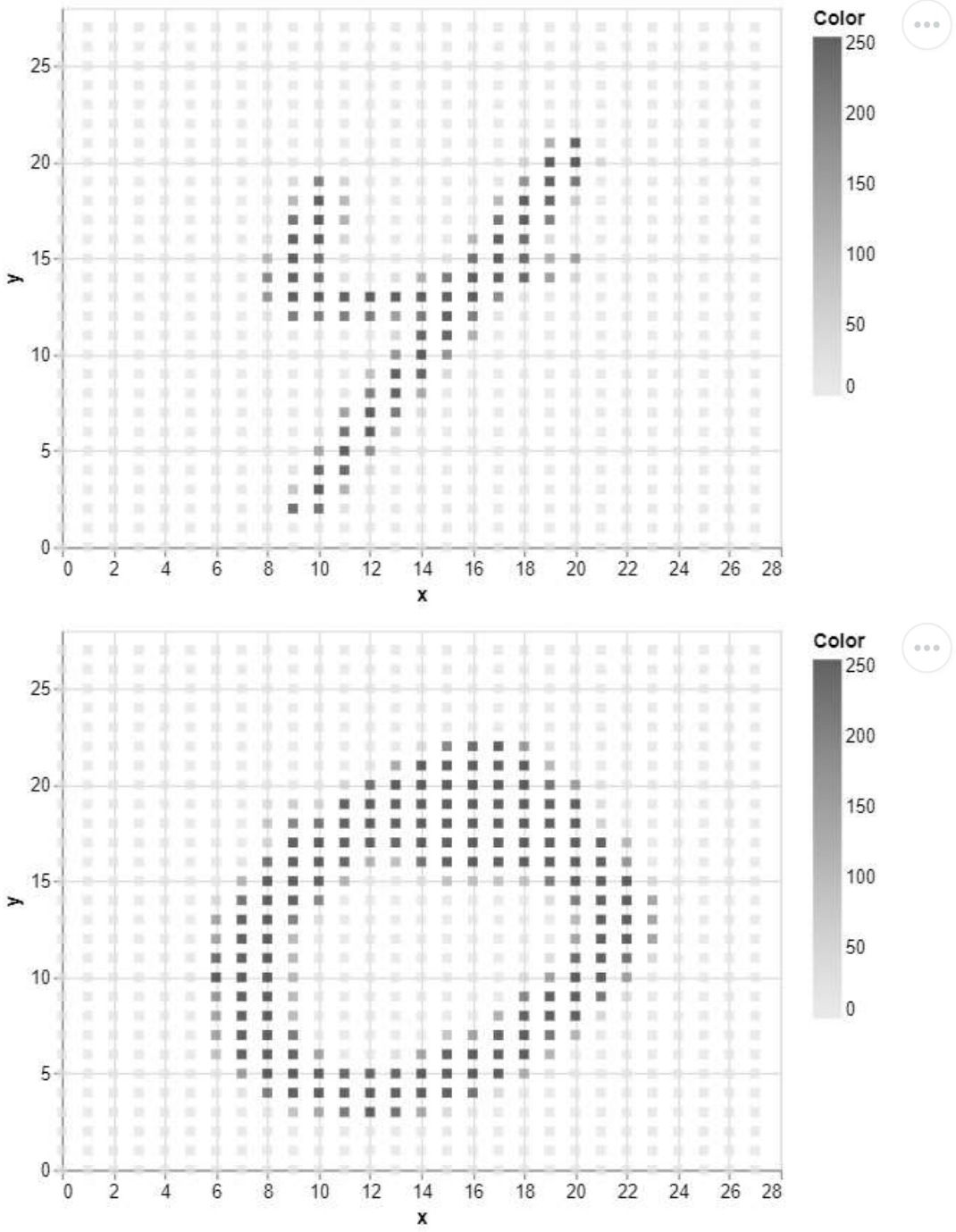


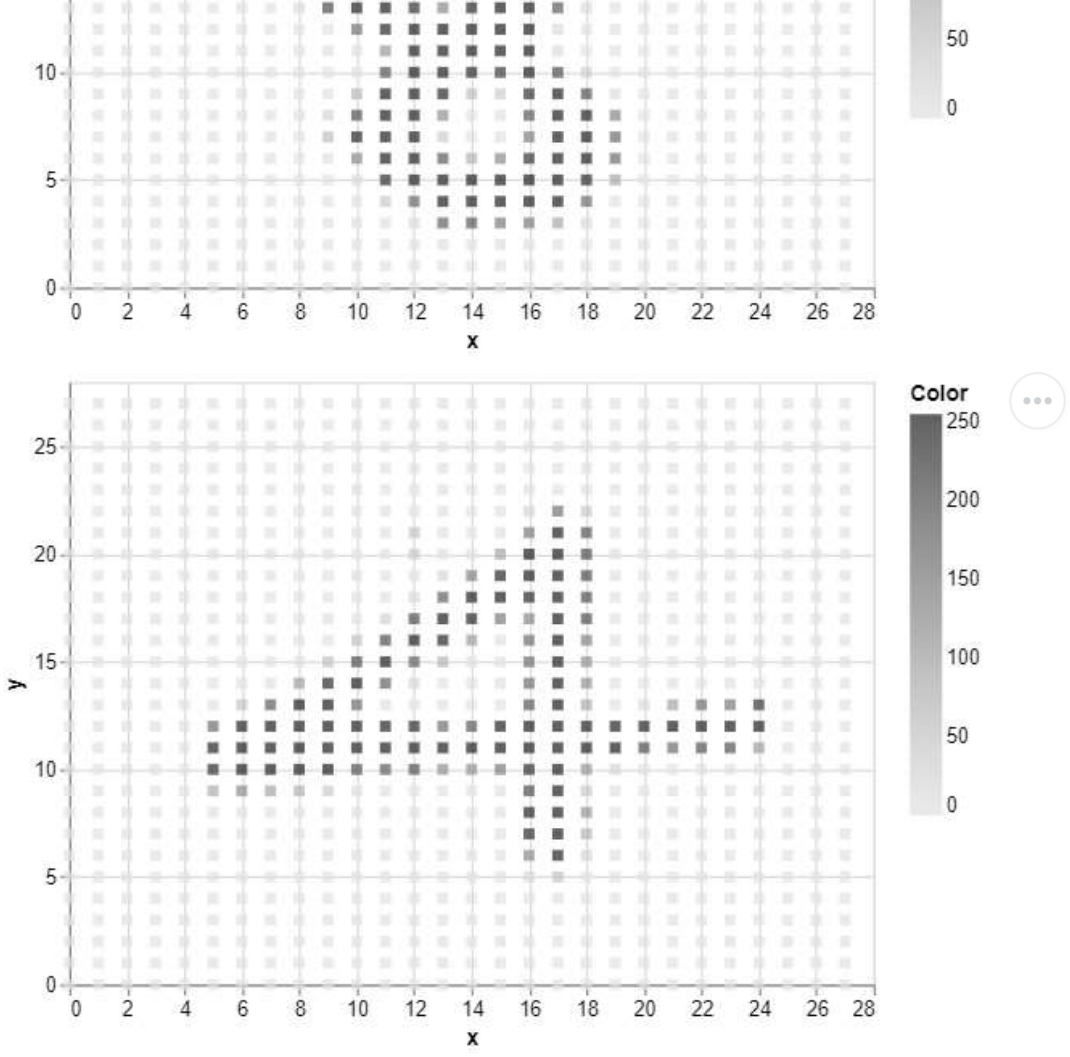



Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
5
15
25


Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
5
15
25

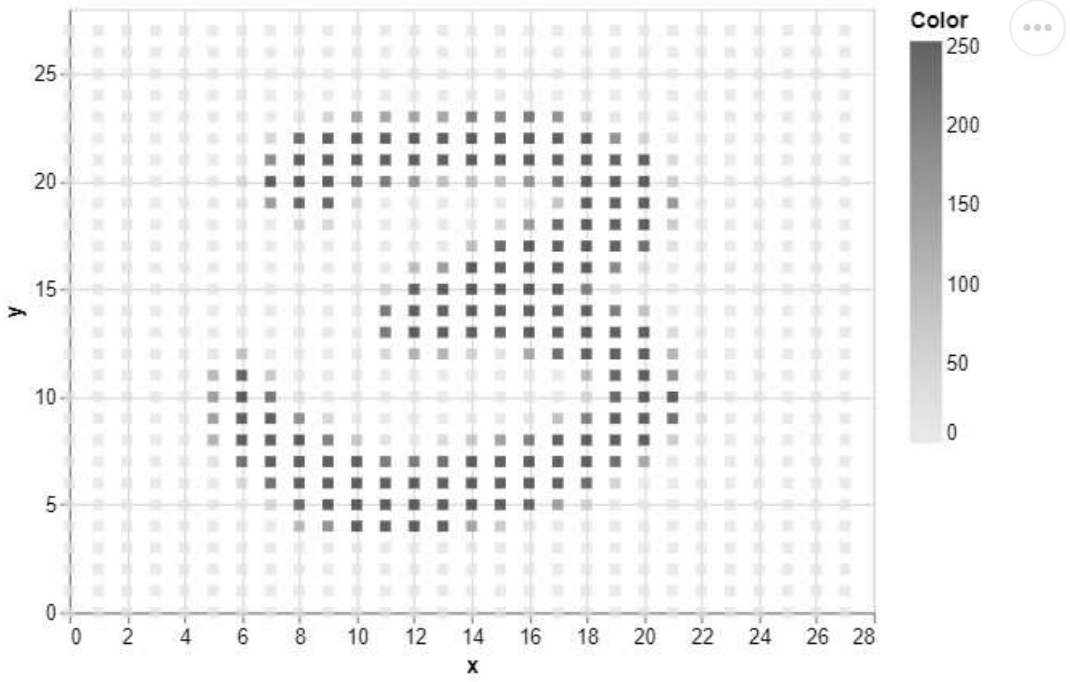

Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
5
15
25

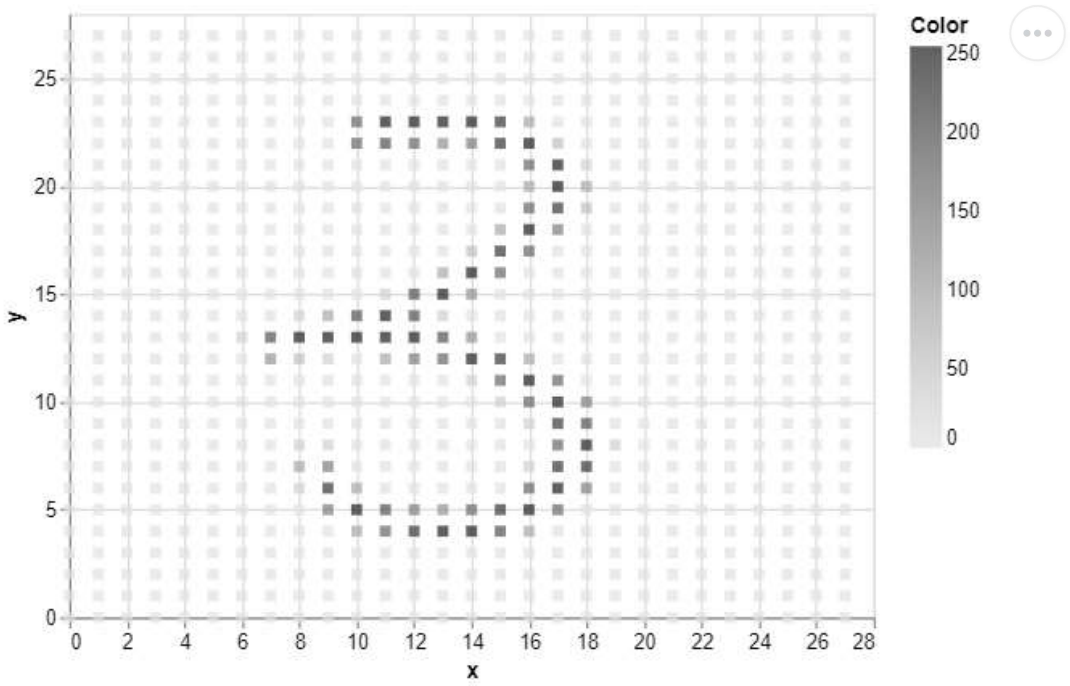

Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
0
5
10
15
20
25

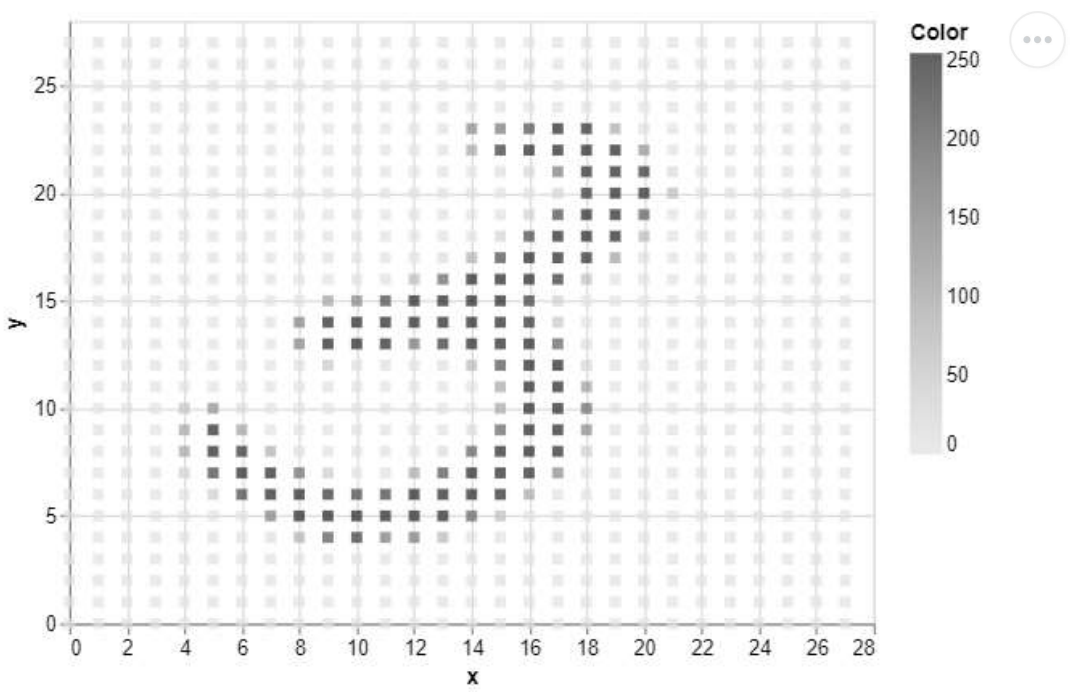

Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
0
5
10
15
20
25

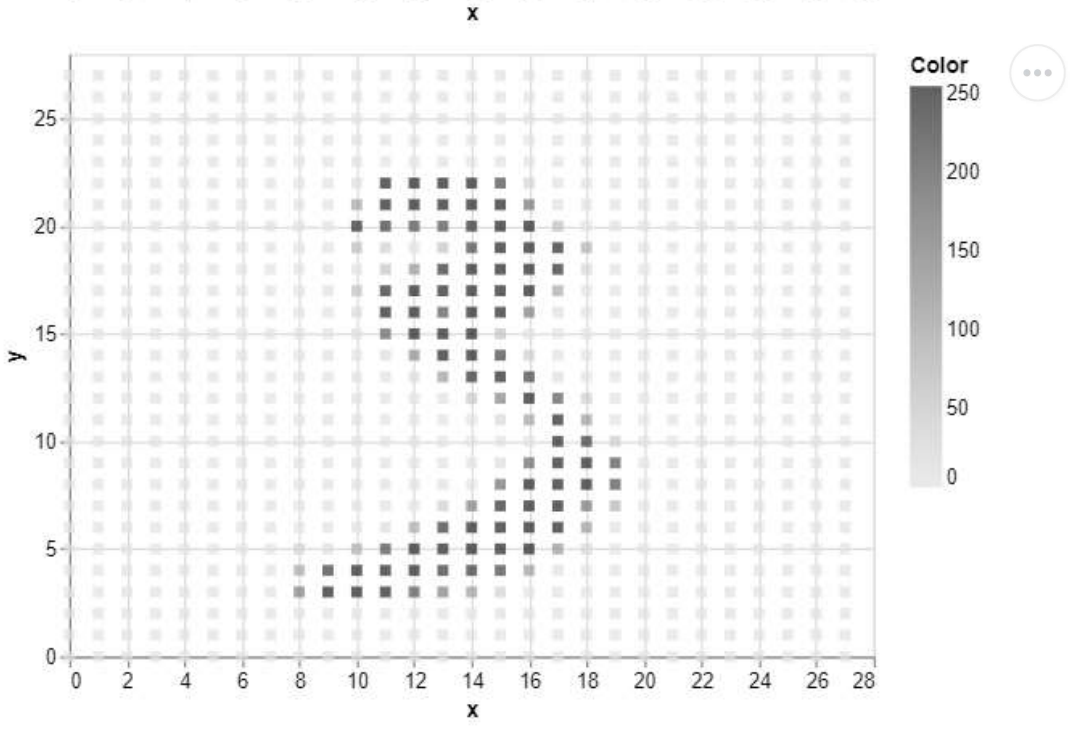
Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
0
5
10
15
20
25

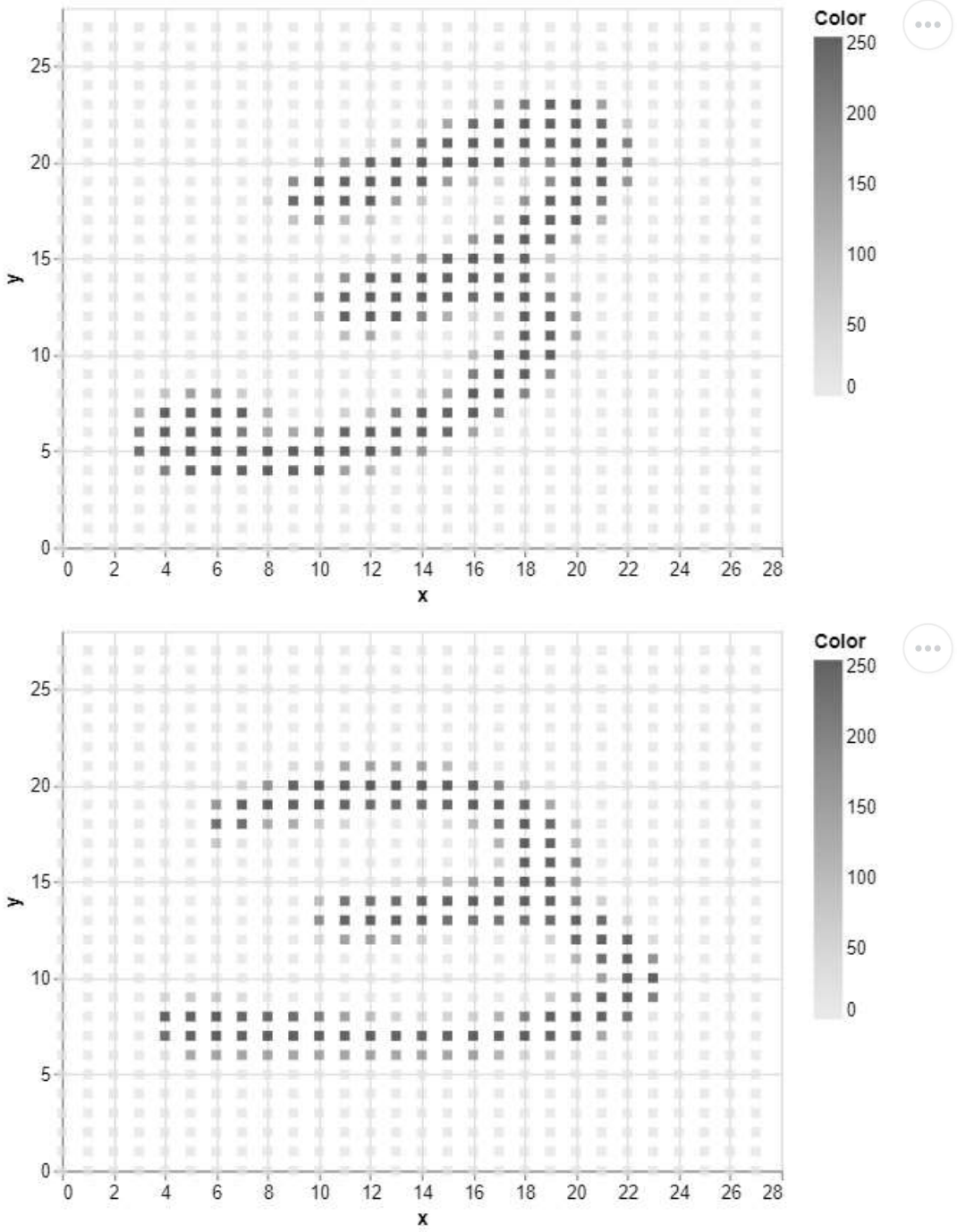
Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
0
5
10
15
20
25

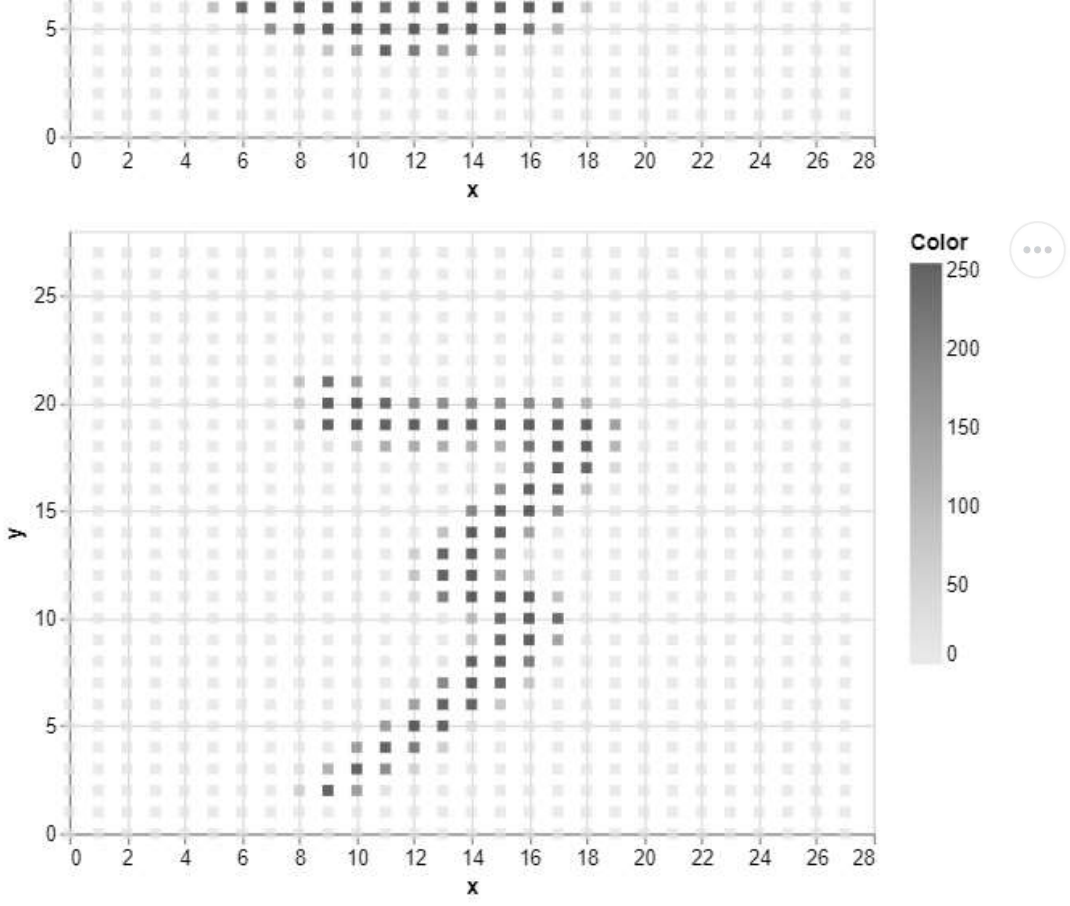

Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
5
15
25


Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
5
15
25


Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
5
15
25

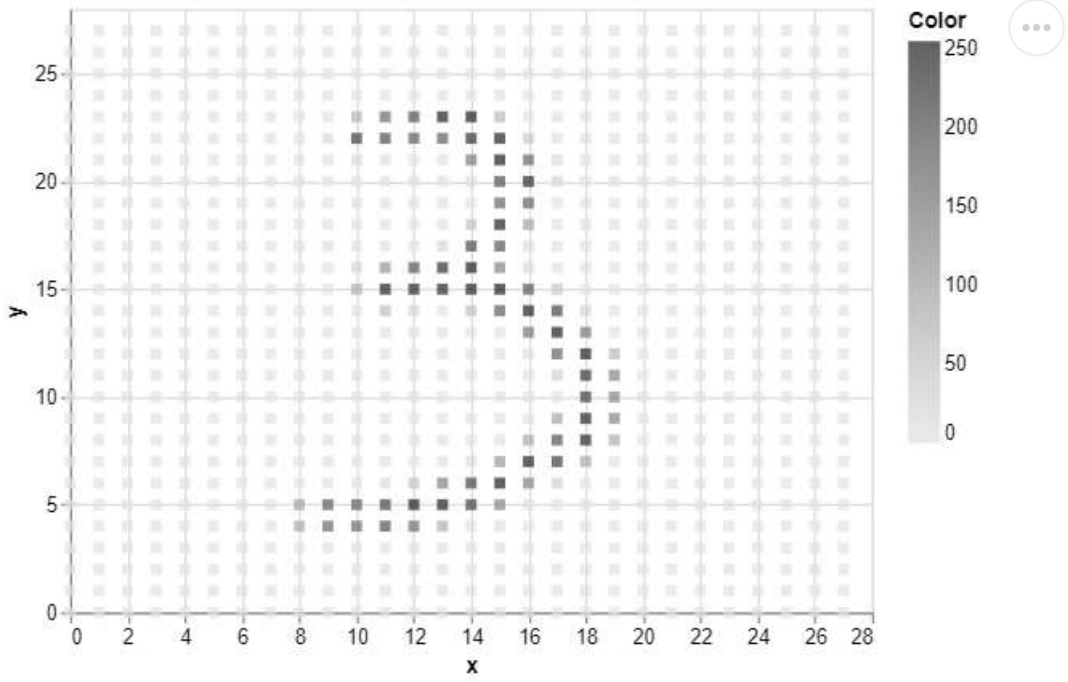

Color
250
200
150
100
50
0
y
x
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
5
15
25

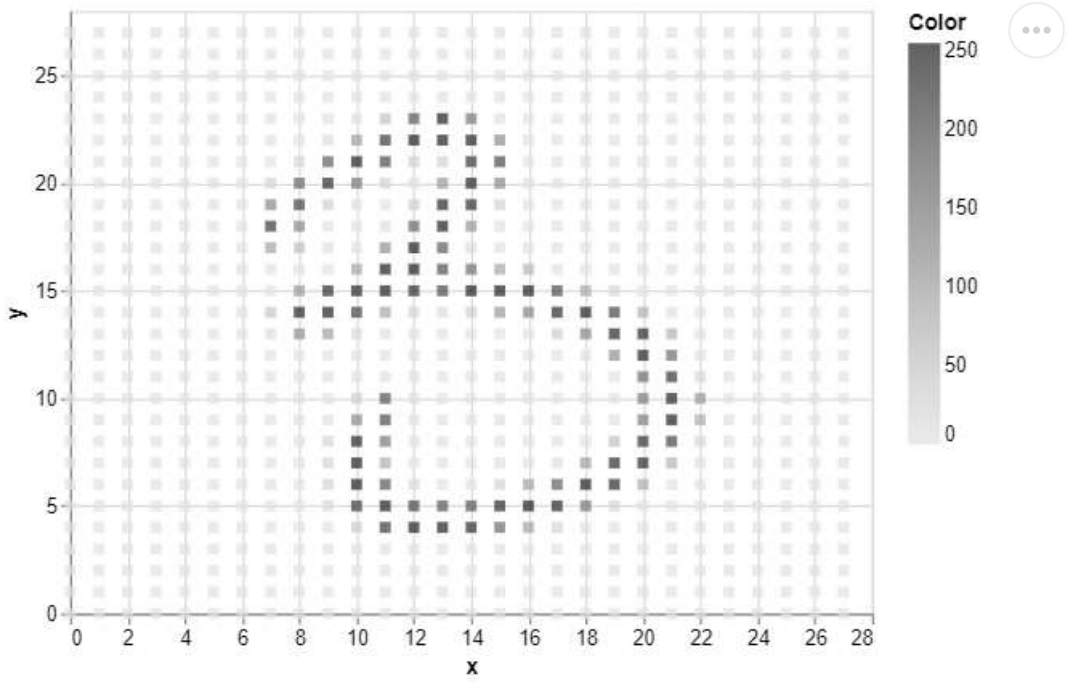

Color
250
200
150
100
50
0
y
x
0
5
10
15
20
25
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28


Color
250
200
150
100
50
0
y
x
0
5
10
15
20
25
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28

Color
250
200
150
100
50
0
y
x
0
5
10
15
20
25
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28

Color
250
200
150
100
50
0
y
x
0
5
10
15
20
25
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28





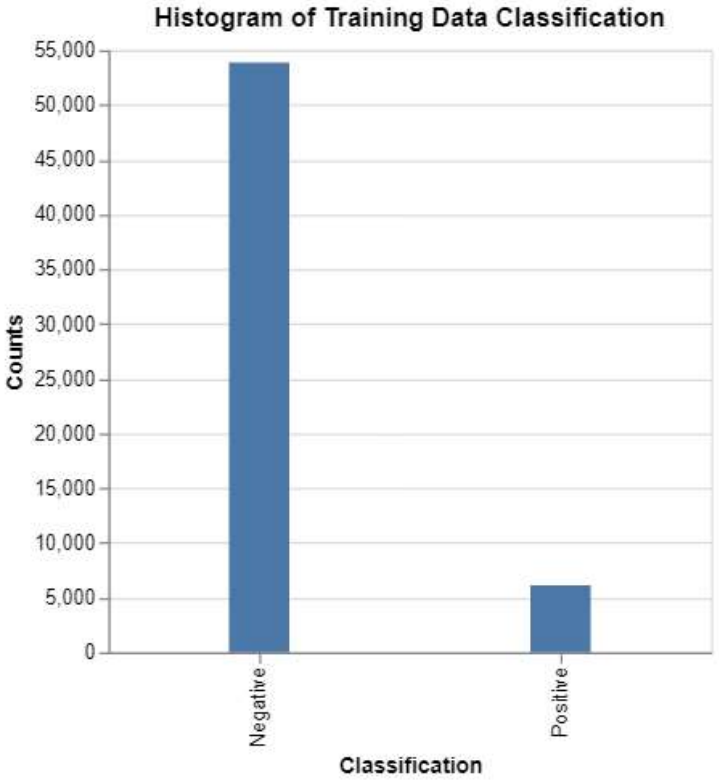
(b) Evaluating Data Balance

In [ ]:
```
print(f"""There are {(y_train == 1).sum(): ,} positive samples in the training data, \
which is {(y_train == 1).sum() / y_train.shape[0] * 100: .2f}% of the training data.""")
```

```
There are  6,129 positive samples in the training data, which is  10.21% of the training data.
```

In [ ]:
```
%%capture --no-display
alt.Chart(
    pd.DataFrame(
        {'Counts':
            [(y_train == 1).sum(),
            (y_train == 0).sum()],
        "Classification":
            ['Positive',
            'Negative']}),
        title = 'Histogram of Training Data Classification').mark_bar(size = 30).encode(
                x = 'Classification',
                y = 'Counts').properties(width = 300)
```

Out[ ]:



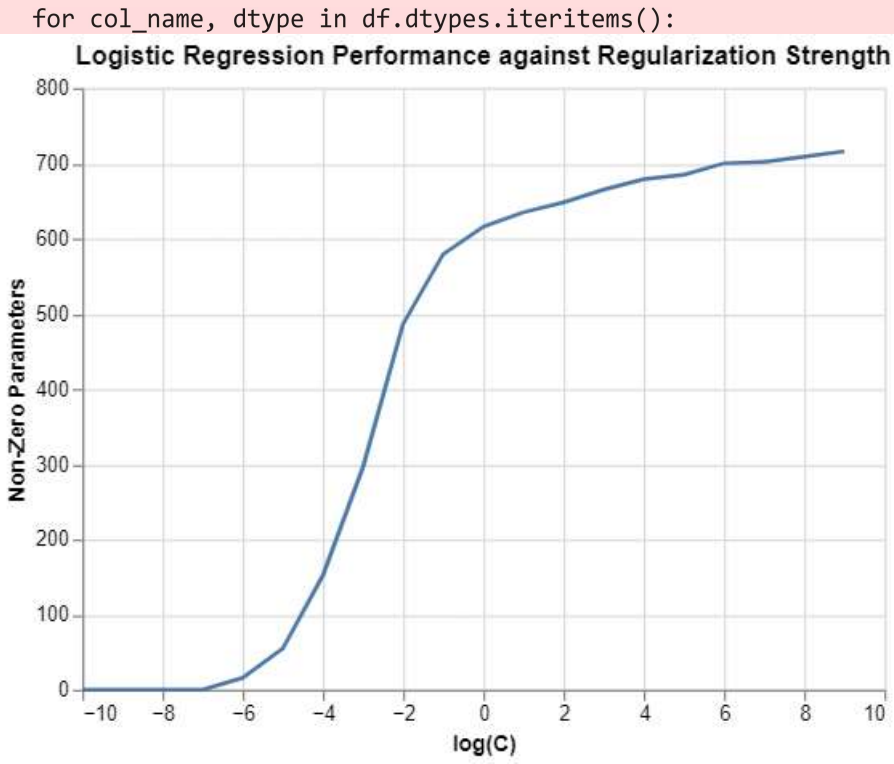
As seen by the plot and the proportion of data which is positive, the data is substantially unbalanced, which will likely lead to over-estimating the classes as negative, because, simply by classifying all data as negative, a model can have ~ 90% accuracy.

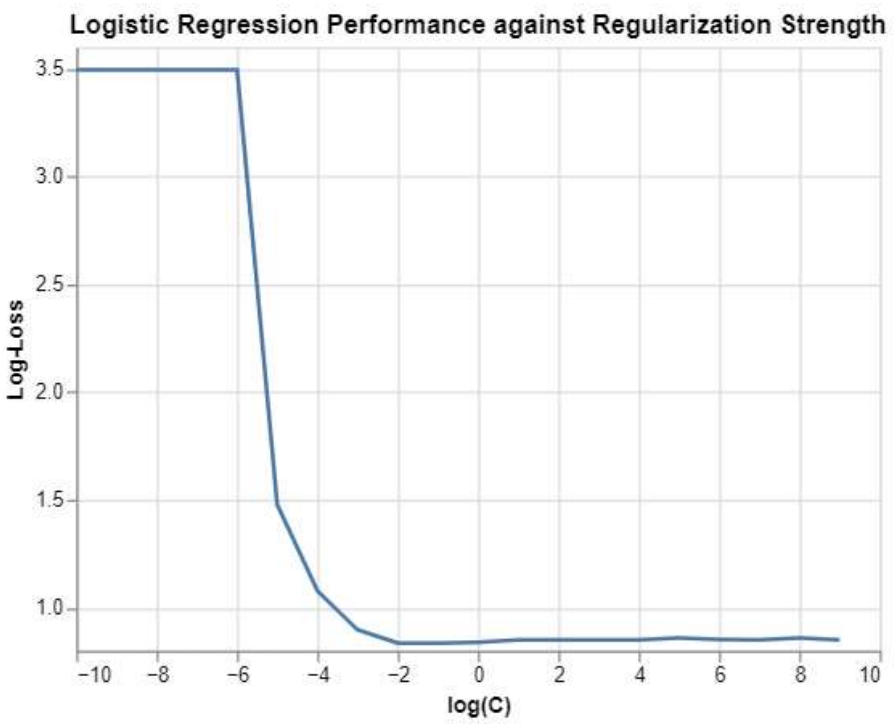
(c) Analyzing Regularization for the Model

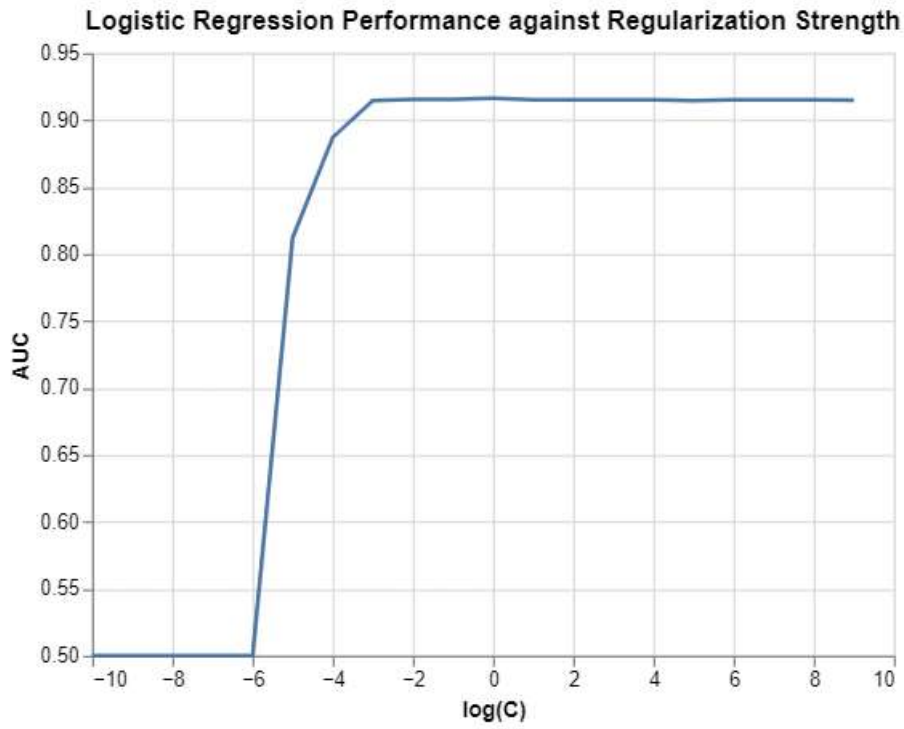
In [ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss, f1_score, roc_auc_score
```

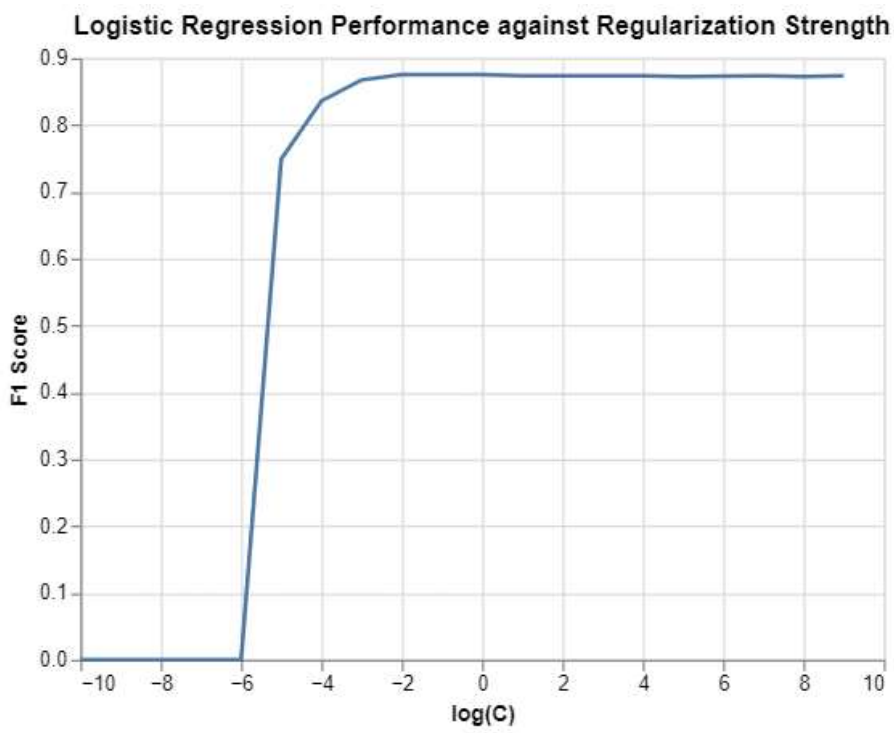
In [ ]:
```
# Create a list of C values to test
C = [c for c in range(
    np.log(1e-4).astype('int') - 1,
    np.log(1e4).astype('int') + 1,
    round((np.log(1e4).astype('int') - np.log(1e-4).astype('int')) / 20))]

# Create a table to store the results
table = pd.DataFrame(
    index = C,
    columns = [
        'Non-Zero Parameters',
        'Log-Loss',
        'AUC',
        'F1 Score'])

# Loop through the C values, fit the model, and track the results
for c in C:
    sklearn_logmodel = LogisticRegression(
        penalty = 'l1',
        solver = 'liblinear',
        max_iter = 5000,
        C = 10 ** c)

    sklearn_logmodel.fit(X_train, y_train)
    table.loc[c, 'Non-Zero Parameters'] = (sklearn_logmodel.coef_ != 0).sum()
    table.loc[c, 'Log-Loss'] = log_loss(y_test, sklearn_logmodel.predict(X_test))
    table.loc[c, 'AUC'] = roc_auc_score(y_test, sklearn_logmodel.predict(X_test))
    table.loc[c, 'F1 Score'] = f1_score(y_test, sklearn_logmodel.predict(X_test))


plots = []

# Loop through the columns and create a plot for each
for eachColumn in table.columns:
    plots.append(
        alt.Chart(
            table.reset_index(names = ['log(C)']),
            title = 'Logistic Regression Performance against Regularization Strength'
            ).mark_line().encode(
                x = 'log(C)',
                y = alt.Y(
                    eachColumn,
                    scale = alt.Scale(zero = False))))

for eachPlot in plots:
    display(eachPlot)
```

```
c:\Users\nicho\miniconda3\lib\site-packages\altair\utils\core.py:317: FutureWarning: iteritems is deprecated and will be removed in a future version. Use .items instead.
  for col_name, dtype in df.dtypes.iteritems():
```









The optimal regualarization value for C appears to be .01.

In [ ]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier
```

In [ ]:
```
# Define the four models
sklearn_logmodel1 = LogisticRegression(
    penalty = 'l1',
    solver = 'liblinear',
    max_iter = 5000,
    C = .01)

sklearn_logmodel2 = LogisticRegression(
    penalty = 'l1',
    solver = 'liblinear',
    max_iter = 5000,
    C = 1e100)

LDAmodel = LinearDiscriminantAnalysis()
RFmodel = RandomForestClassifier()
```

In [ ]:
```
%%capture --no-display

# Fit the models
sklearn_logmodel1.fit(X_train, y_train)
sklearn_logmodel2.fit(X_train, y_train)
LDAmodel.fit(X_train, y_train)
RFmodel.fit(X_train, y_train)
```

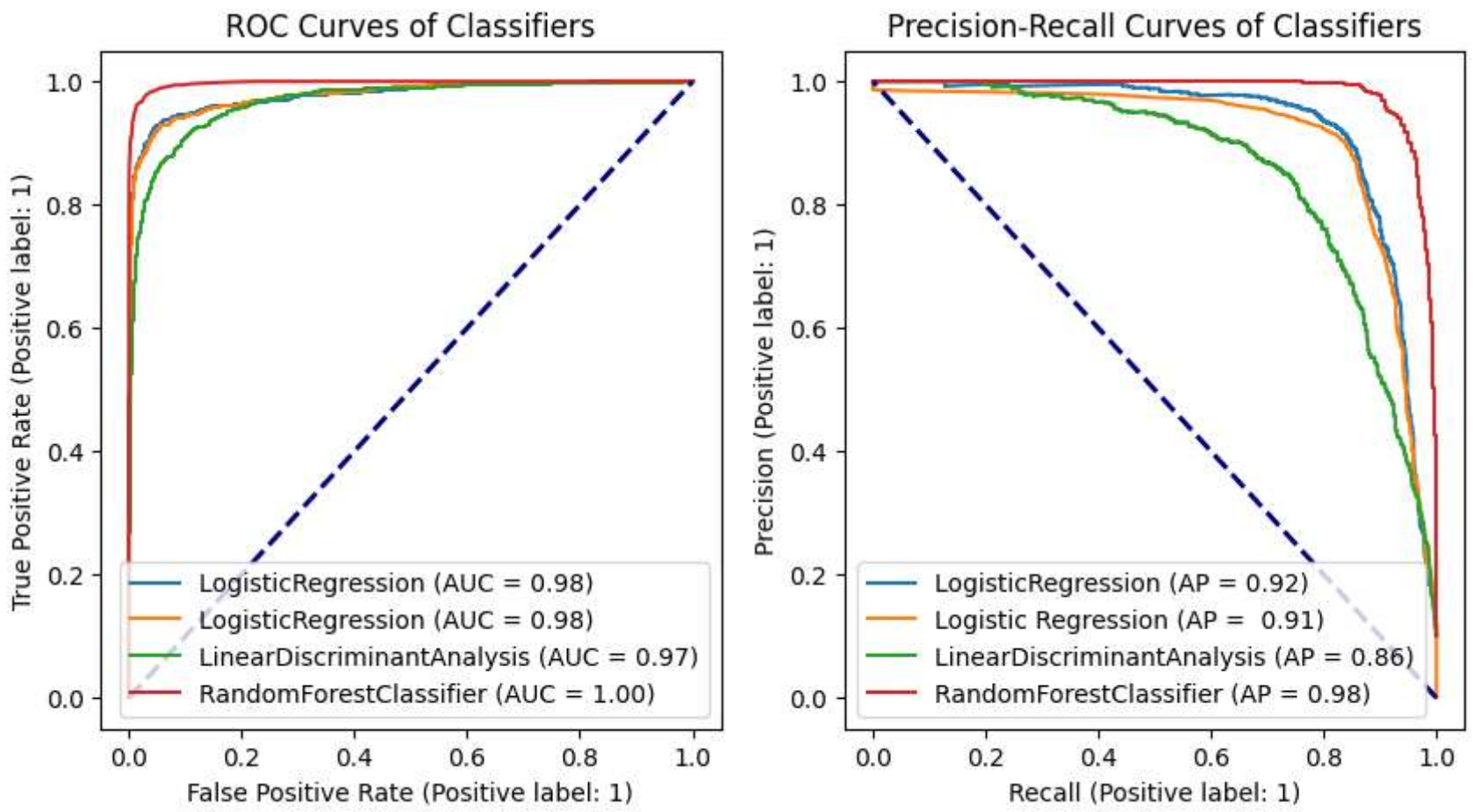
Out[ ]:
```
▾ RandomForestClassifier

RandomForestClassifier()
```

In [ ]:
```
from sklearn.metrics import RocCurveDisplay
from sklearn.metrics import PrecisionRecallDisplay
```

In [ ]:
```
from sklearn.metrics import auc, precision_recall_curve as prcurve

# Setting up plots
fig, axs = plt.subplots(1, 2, figsize = (10, 5))

# Plotting ROC Curves
RocCurveDisplay.from_estimator(sklearn_logmodel1, X_test, y_test, ax = axs[0])
RocCurveDisplay.from_estimator(sklearn_logmodel2, X_test, y_test, ax = axs[0])
RocCurveDisplay.from_estimator(LDAmodel, X_test, y_test, ax = axs[0])
RocCurveDisplay.from_estimator(RFmodel, X_test, y_test, ax = axs[0])

# Overlaying Random Guessing
axs[0].plot([0, 1], [0, 1], color = 'navy', lw = 2, linestyle = '--', label = 'Random Guessing')
```

```
axs[0].set_title("ROC Curves of Classifiers")

# Plotting Precision-Recall Curves
PrecisionRecallDisplay.from_estimator(sklearn_logmodel1, X_test, y_test, ax = axs[1])

precision, recall, thresholds = prcurve(y_test, sklearn_logmodel2.predict_proba(X_test)[:, 1])
logauc = auc(recall, precision)
axs[1].plot(
    sorted(recall),
    sorted(precision, reverse = True),
    label = f'Logistic Regression (AP = {logauc: .2f})')

PrecisionRecallDisplay.from_estimator(LDAmodel, X_test, y_test, ax = axs[1])
PrecisionRecallDisplay.from_estimator(RFmodel, X_test, y_test, ax = axs[1])
axs[1].plot([1, 0], [0, 1], color = 'navy', lw = 2, linestyle = '--')

axs[1].set_title("Precision-Recall Curves of Classifiers")
```

Out[ ]: Text(0.5, 1.0, 'Precision-Recall Curves of Classifiers')



Regularization did not seem to make a substantial difference on the Logistic Regression model's ability to generalize to unseen data in this data set. For this data, the Random Forest Classifier performed the best on the test data, and therefore is the model that I would recommend using for this data set.