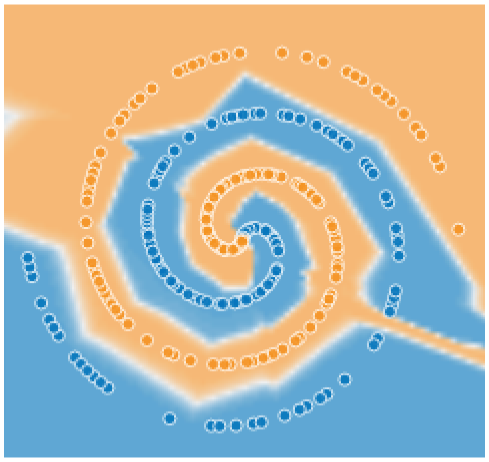
Lab 6 - Deep Learning 1

The goal of this week's lab is to learn what a deep neural network is and how it relates to previous linear models. We are going to build networks today that can automatically learn to adapt to very complex shapes.



In the past two weeks we used a simple linear model for classification. In practice however, many datasets are not linearly separable, i.e., not all classification problems can be solved by a linear classifier, as we saw earlier on the circle dataset.

Deep neural networks provide a way to learn flexible shapes without specifying features. This is achieved by stacking multiple layers on top of each others.

Deep neural networks form the backbone of deep learning approaches, which have seen tremendous successes in applications such as machine translation, document summarization, image classification, and speech recognition.

Here is a video of a neural network generating faces.

[Face Generation]

This week we will walk through the basics of deep neural networks.

- **Review**: Linear Classifiers and Features
- **Unit A**: TensorFlow, Training Linear Classifiers
- **Unit B**: Neural Networks

Review

Last time we trained a linear classifier for binary classification.

```
import altair as alt
import pandas as pd
import sklearn.linear_model
```

We will also turn off warnings.

```
import warnings
warnings.filterwarnings('ignore')
```

```
# Step 1. Create out data.
df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/circle.csv")
df_train = df.loc[df["split"] == "train"]
df_test = df.loc[df["split"] == "test"]
```

Step 2. Create a linear model and fit it to data.

```
model = sklearn.linear_model.LogisticRegression()
model.fit(X=df_train[["feature1", "feature2"]],
          y=df_train["class"])
```

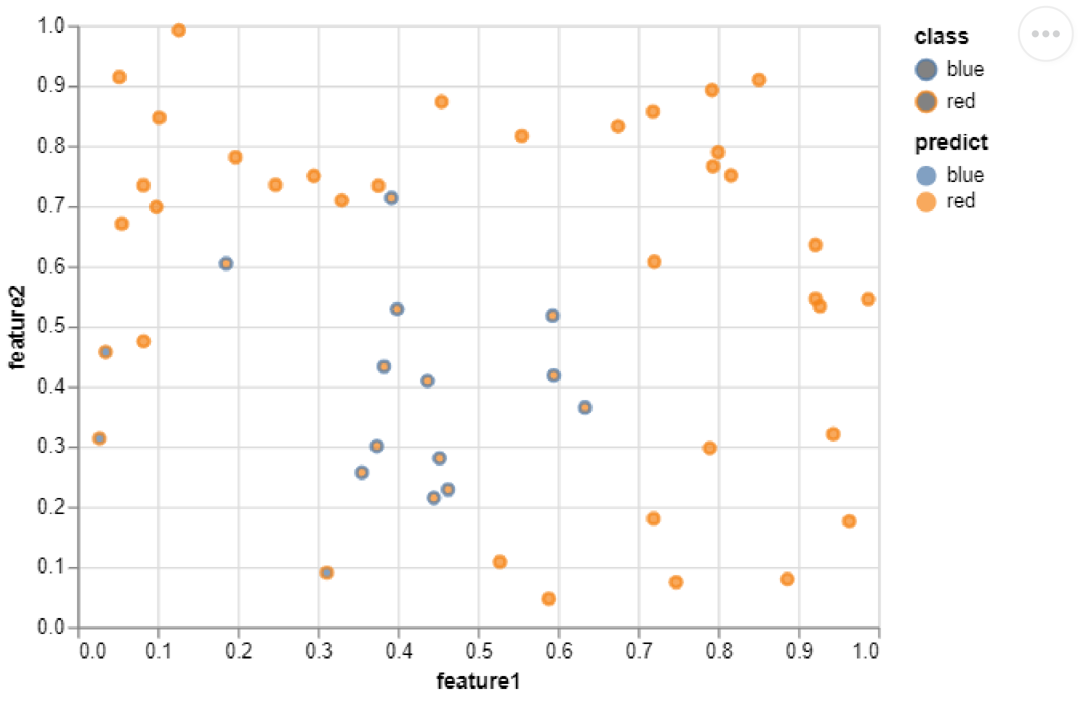
```
     LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                        intercept_scaling=1, l1_ratio=None, max_iter=100,
                        multi_class='auto', n_jobs=None, penalty='l2',
                        random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                        warm_start=False)
```

Step 3. Predict.

```
df_test["predict"] = model.predict(df_test[["feature1", "feature2"]])
```

We can see that the linear classifier fails to classify some data points due to its being linear.

```
correct = (df_test["predict"] ==  df_test["class"])
df_test["correct"] = correct
chart = (alt.Chart(df_test)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color = "class",
        fill = "predict",
        tooltip = ["correct"]
    ))
chart
```



We can improve on this approach by adding a new feature column.

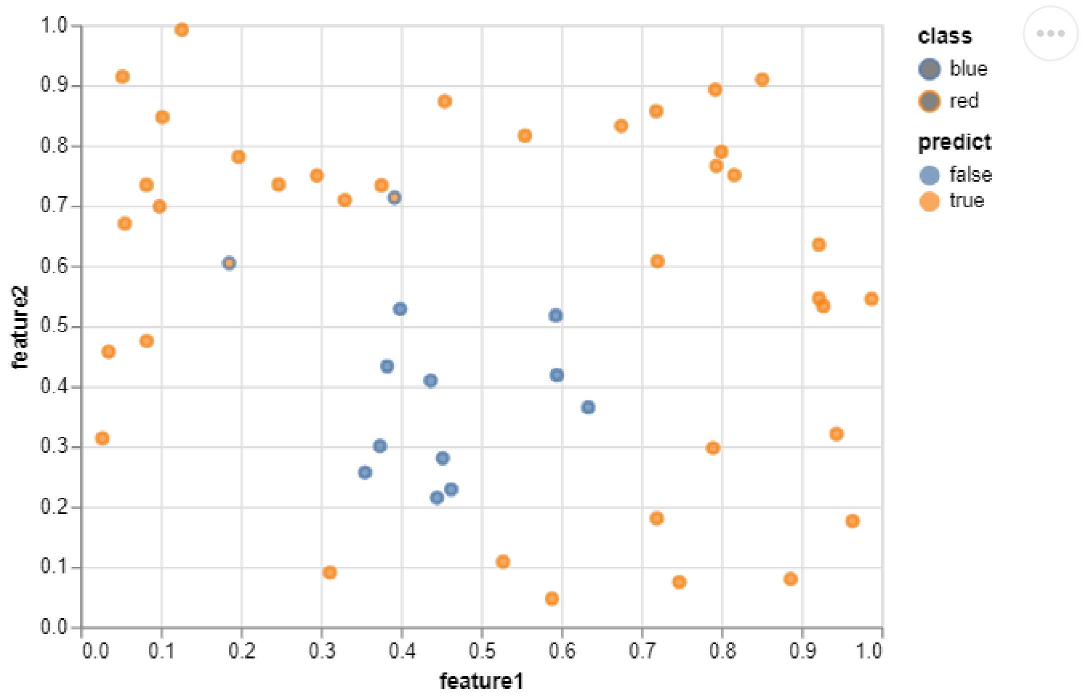
```
def mkdistance(row):
    f1 = row["feature1"] - 0.5
    f2 = row["feature2"] - 0.5
    return f1*f1 + f2*f2
```

```
# We add it to the model with either `map` or `apply`.
df_train["feature3"] = df_train.apply(mkdistance, axis=1)
df_test["feature3"] = df_test.apply(mkdistance, axis=1)
```

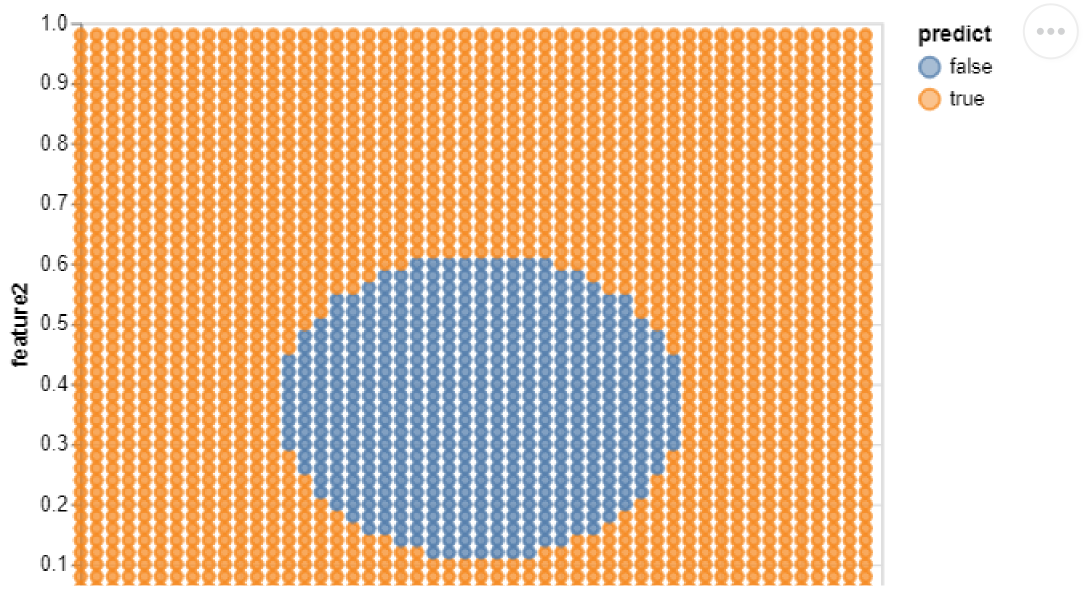
If instead we fit with the new features

```
model.fit(X=df_train[["feature2", "feature3"]],
          y=df_train["class"] == "red")
```

```
df_test["predict"] = model.predict(df_test[["feature2", "feature3"]])


chart = (alt.Chart(df_test)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color = "class",
        fill = "predict",
        tooltip = ["correct"]
    ))
chart
```

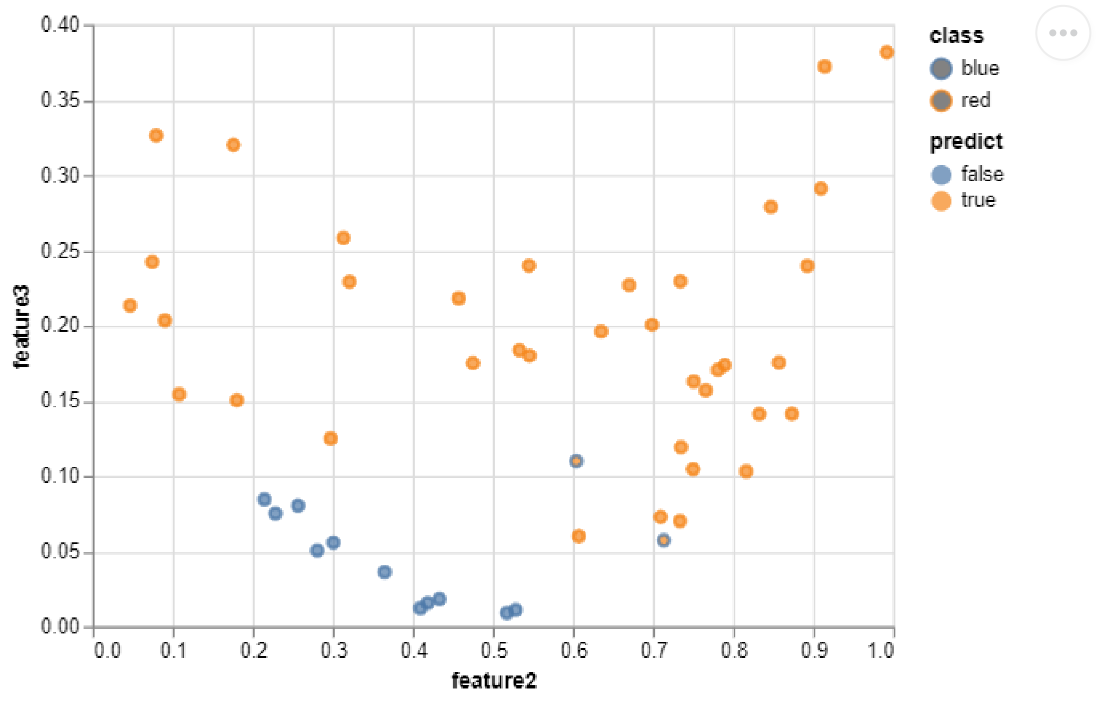


```
# We can also check the behavior of our classifier on all possible inputs.
all_df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/all_points.csv")
all_df["feature3"] = all_df.apply(mkdistance, axis=1)
all_df["predict"] = model.predict(all_df[["feature2", "feature3"]])
chart = (alt.Chart(all_df)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color="predict",
        fill = "predict",
    ))
chart
```



Remember though this is because we changed the features. The model is still a linear seperator in the new features.

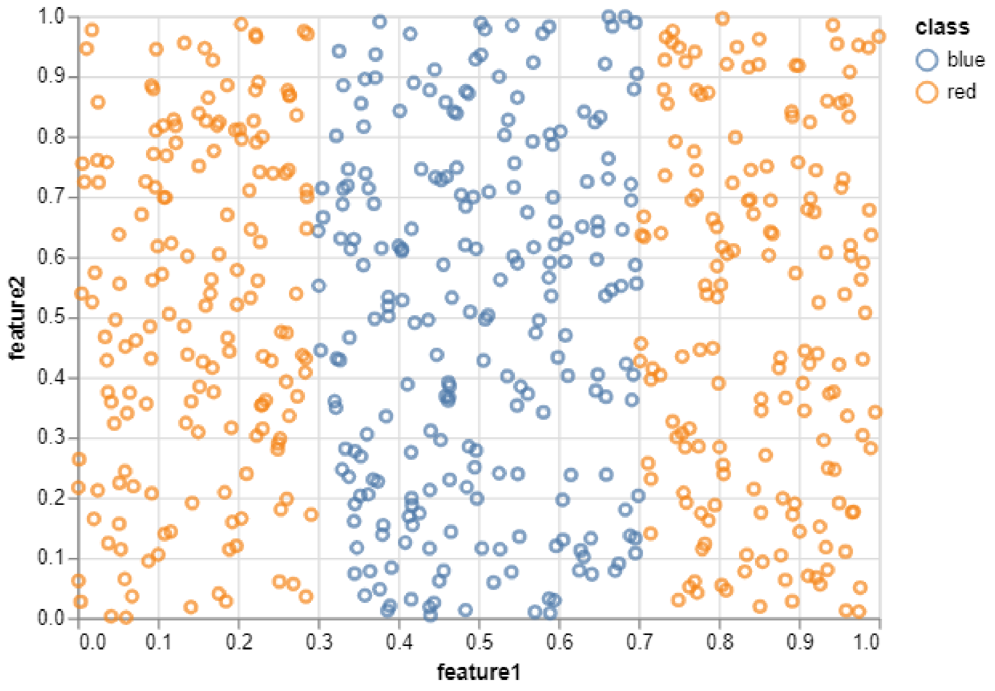
```
chart = (alt.Chart(df_test)
    .mark_point()
    .encode(
        x = "feature2",
        y = "feature3",
        color = "class",
        fill = "predict",
        tooltip = ["correct"]
    ))
chart
```



We can do the same thing for other datasets.

```
df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/center.csv")
```

```
chart = (alt.Chart(df)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color = "class"
    ))
chart
```



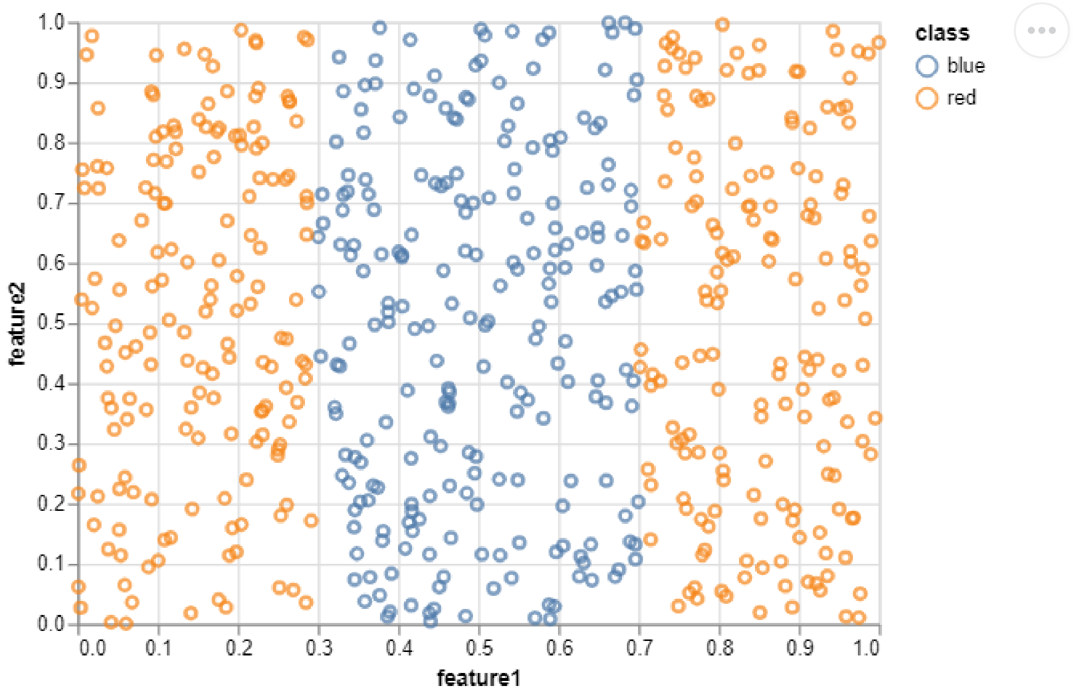
Review Exercise

Add new features and make predictions on this df dataset.

```
#📝📝📝📝 FILLME
def mkfeature(row):
    return abs(row["feature1"] - 0.5)
df["feature3"] = df.apply(mkfeature, axis=1)
model.fit(X=df[["feature1", "feature2", "feature3"]],
          y=df["class"] == "red")
df["predict"] = model.predict(df[["feature1", "feature2", "feature3"]])
```

Graph the result

```
chart = (alt.Chart(df)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color = "class"
    ))
chart
```



📝📝📝📝 FILLME

Unit A

Linear Models in TensorFlow

Linear classifiers can only produce linear "decision boundaries", i.e., there is a line such that points on one side of the line are classified as one class, whereas points on the other side are classified as the other class.

Last class we got around this issue by telling the ML system what shapes we wanted. However this required us to know enough about the problem to specify what these were.

For today's class we are going to extend our model such that it can produce more general shapes without requiring us to tell it what they are.

We will do this with a library known as TensorFlow. TensorFlow is the main library that companies doing machine learning use. When people talk about AI these days they are often talking about machine learning with TensorFlow.

To get started we will need these imports. We will use TensorFlow through a library known as Keras.

```
import tensorflow as tf
import keras
from keras.wrappers.scikit_learn import KerasClassifier
from keras.models import Sequential
from keras.layers import Dense, Activation
```

The first thing we will do is rebuild our LinearClassifier in Keras.

We also need a function to create model, which is required for building a general `KerasClassifier`.

```
def create_model(rate=1.0):
    # Makes it the same for everyone in class
    tf.random.set_seed(2)

    # Create model
    model = Sequential()
    model.add(Dense(1, activation="sigmoid")) # Linear Classifier

    # Compile model
    optimizer = tf.keras.optimizers.SGD(
        learning_rate=rate
    )
    model.compile(loss="binary_crossentropy",
                  optimizer=optimizer,
                  metrics=["accuracy"])
    return model
```

Next, we use `KerasClassifier` to turn a TensorFlow model into one that can be used by Scikit Learn. We have to give it a couple of arguments to get it started.

```
model = KerasClassifier(build_fn=create_model,
                        epochs=10,
                        batch_size=20,
                        verbose=False)
```

Lastly, we fit the classifier on training data and apply it to all points to check the shape of its decision boundary.

Load the data.

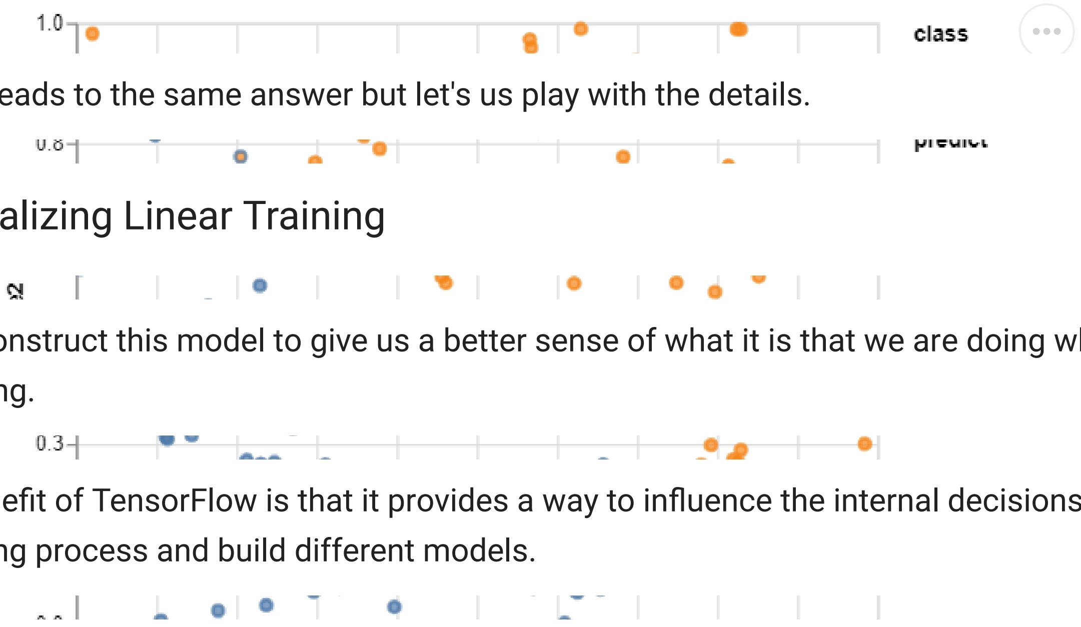
```
df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/simple.csv")
```

This part is the same as Scikit-Learn.

```
model.fit(x=df[["feature1", "feature2"]],
          y=(df["class"] == "red"))
df["predict"] = model.predict(df[["feature1", "feature2"]])
```

Pretty good classification.

```
chart = (alt.Chart(df)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color="class",
        fill = "predict",
    ))
chart
```



This leads to the same answer but let's us play with the details.

Visualizing Linear Training

We construct this model to give us a better sense of what it is that we are doing when we do linear training.

A benefit of TensorFlow is that it provides a way to influence the internal decisions made during the training process and build different models.

To get a sense how this works, let us look at the TensorFlow playground.

[Tensorflow Playground]

This is a tool that allows us to build different machine learning models and play with them in the browser.

Here is an example that looks like our linear model. Press the `Play` button to run it.

[Example 1]

There are several things to look at in the tool.

1. Try clicking on a different Dataset to see how it is difficult for the linear model to fit it.
2. Try altering the features to see some of the tricks that we used last week (squares and sines) help fix this issue.
3. Look at the `Output` graph above the points on the right. What is that graph showing.

What is training?

Now let us look at what happens when we build a tool like this ourselves. This is the stuff that gets

```
model = create_model()
```

We first make our features and target class. We then convert them to TensorFlow format.

```
X = df[["feature1", "feature2"]]
y = df["class"] == "red"
X = tf.convert_to_tensor(X)
y = tf.convert_to_tensor(y)
```

The key thing a model needs to compute is the `loss`. This tells us how well the model is currently doing.

Roughly the loss is a numerical value that counts how many points are on each side of the line, and how far they are.

This is the graph that we saw in the browser version of the tool.

```
loss = model.compiled_loss(y, model(X))
print(loss)
```

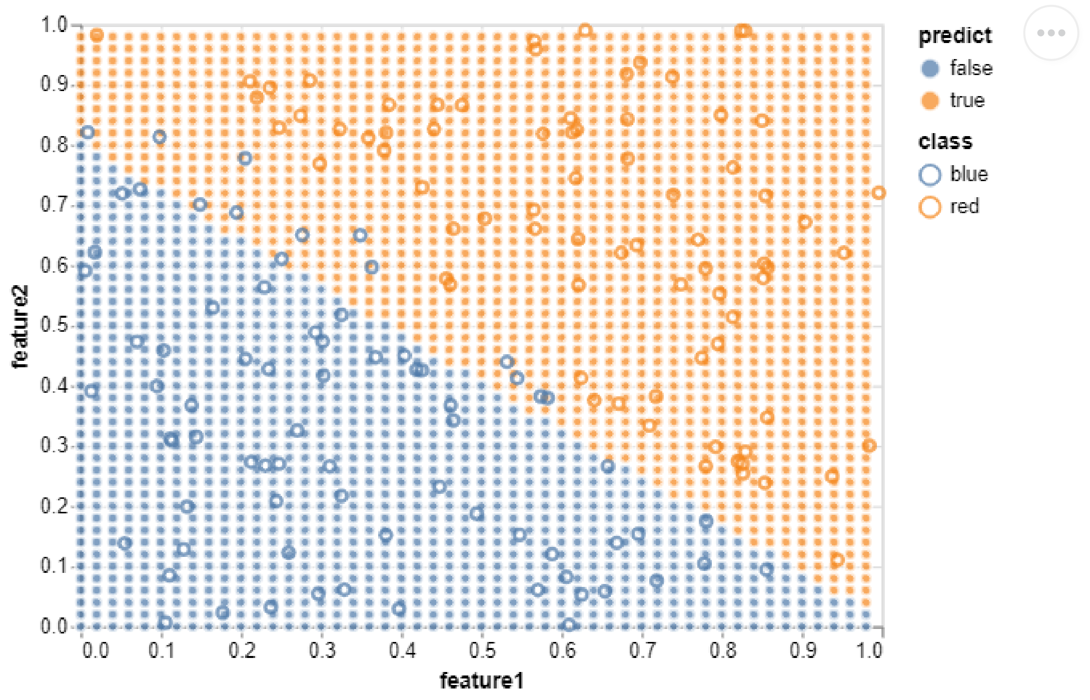
```
     tf.Tensor(0.46146888, shape=(), dtype=float32)
```

If we graph it, we can see that it is not doing well. Many of the points are on the wrong side of the line.

```
all_df["predict"] = model.predict(all_df[["feature1", "feature2"]]) > 0.5
```

```
chart = (alt.Chart(df)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color="class",
    ))
chart2 = (alt.Chart(all_df)
    .mark_point(color="white")
    .encode(
        x = "feature1",
        y = "feature2",
        fill = "predict",
    ))
chart2 + chart
```

```
chart2 + chart
```



Inside the model there are a collections of `parameters`. By adjusting these, we can make the loss go up or down. For example in our model there are 3 trainable parameters.

```
model.summary()
```

```
    Model: "sequential_1"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    dense_1 (Dense)              (None, 1)                 3
    =================================================================
    Total params: 3
    Trainable params: 3
    Non-trainable params: 0
    _________________________________________________________________
```

To improve our model, we need to adjust the line to minimize the loss function. This can be done using basic calculus and derivatives. The system computes derivatives to try to make the loss better.

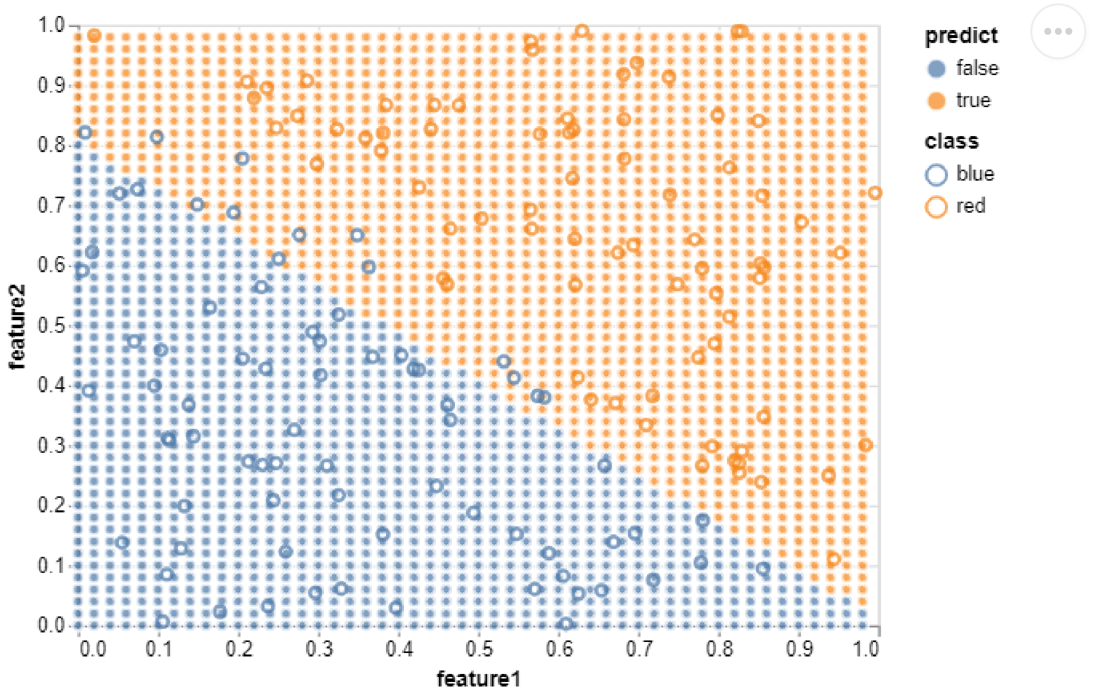
While we don't have time to go into details, you can read more about the gradient descent algorithm that is used.

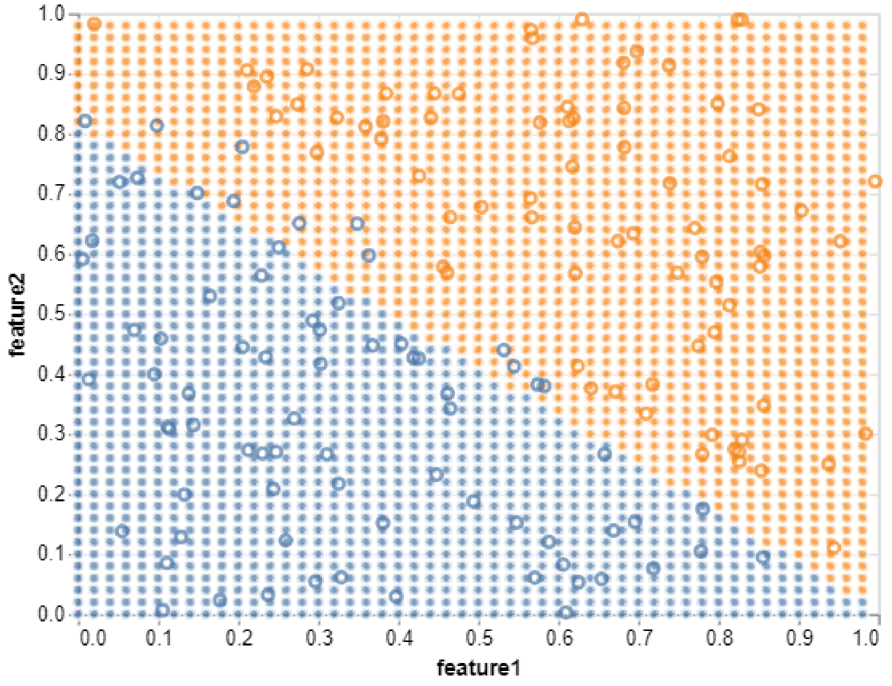
The below code takes one step of gradient descent and adjusts the parameters of the model.

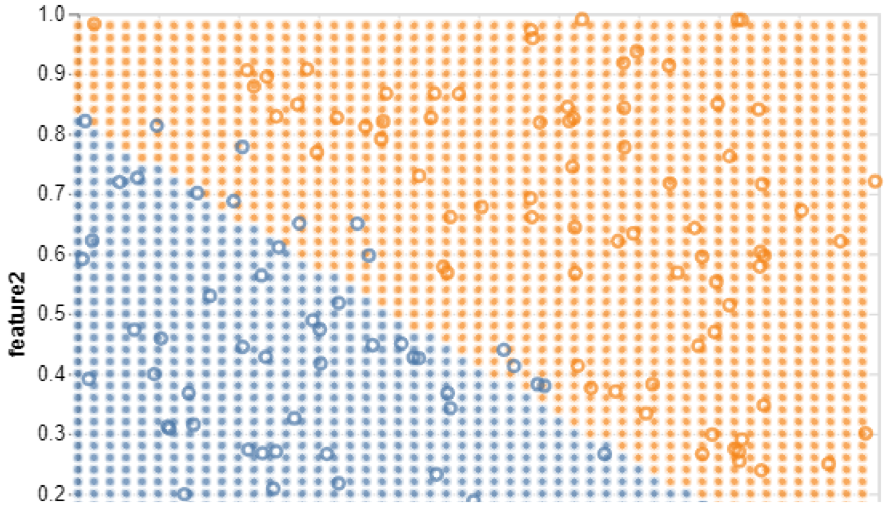
```
pick = []
pick = []
for i in range(50):
    # Compute loss
    with tf.GradientTape() as tape:
        loss = model.compiled_loss(y, model(X, training=True))

    # Compute derivatives
    gradients = tape.gradient(loss, model.trainable_variables)

    # Adjust parameters
    model.optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    # Save for graphing.
    if i % 2 == 0:
        t = all_df.copy()
        t["predict"] = model.predict(all_df[["feature1", "feature2"]]) > 0.5
        pick.append(t)
```

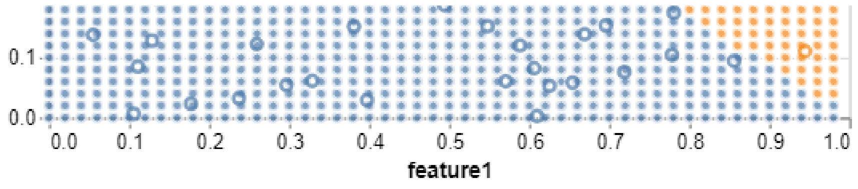
Here we can see all the graphs. See how it gets better each time.

```
vc = alt.vconcat()
for p in pick:
    chart2 = (alt.Chart(p)
              .mark_point(color="white")
              .encode(
                  x = "feature1",
                  y = "feature2",
                  fill = "predict"
              ))
    vc &= (chart2 + chart)
vc
```

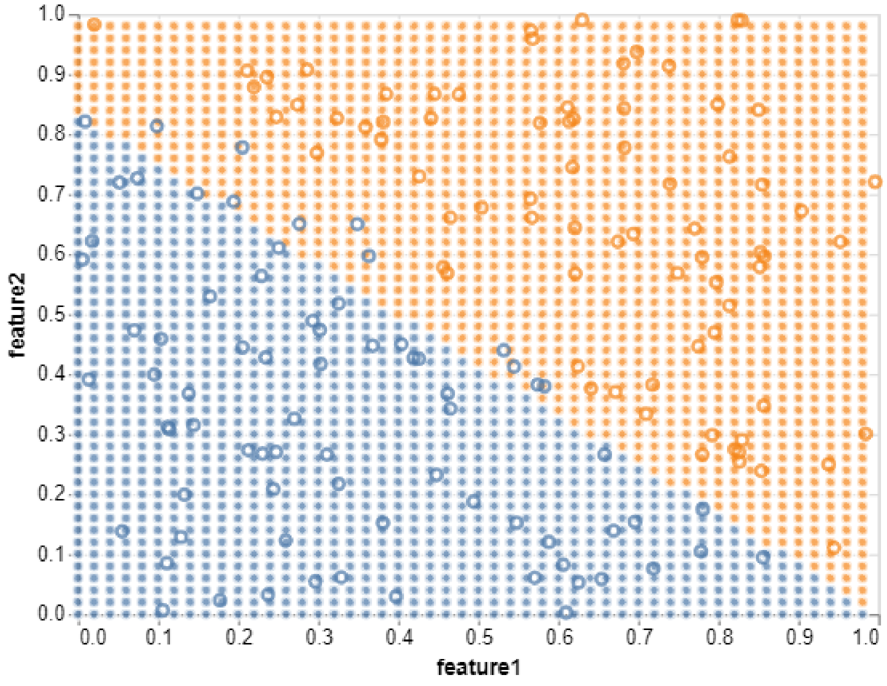

1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
feature2
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1
predict
false
true
class
blue
red


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
feature2
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1

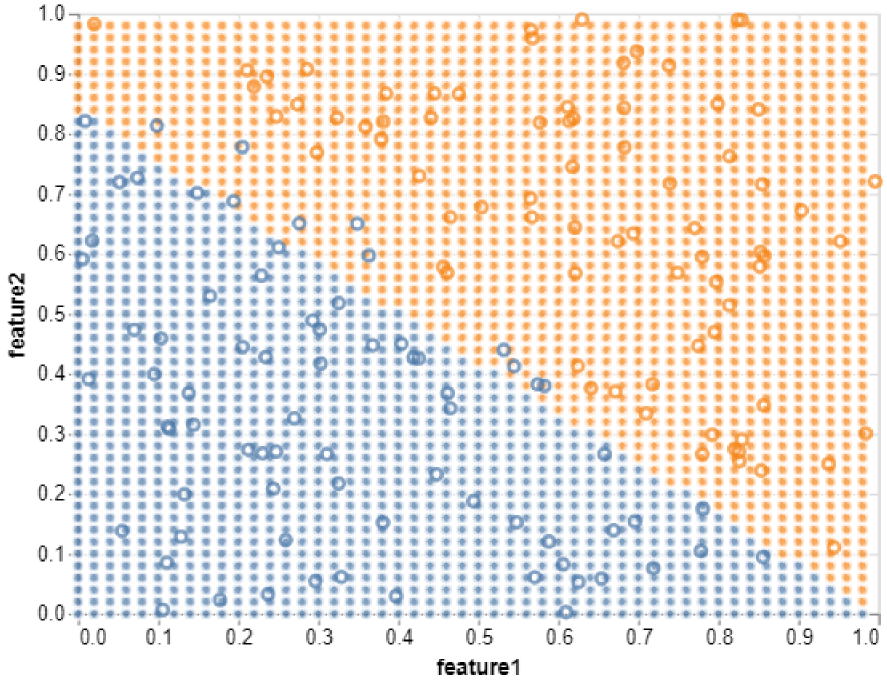

1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
feature2



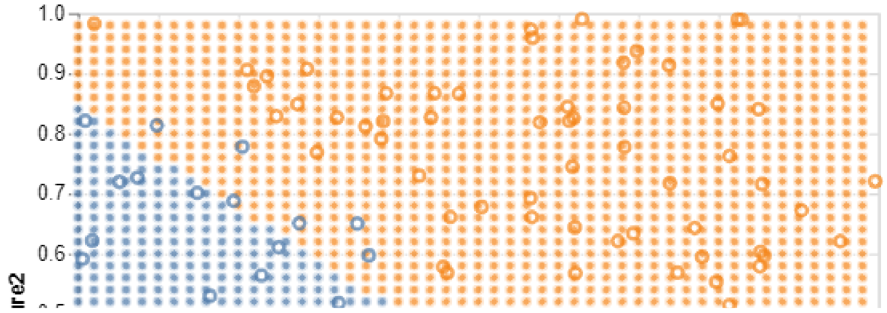
0.1
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1



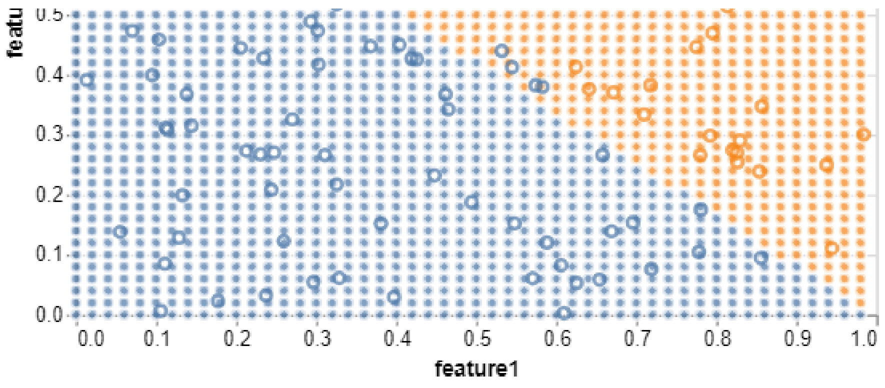
feature2
1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1

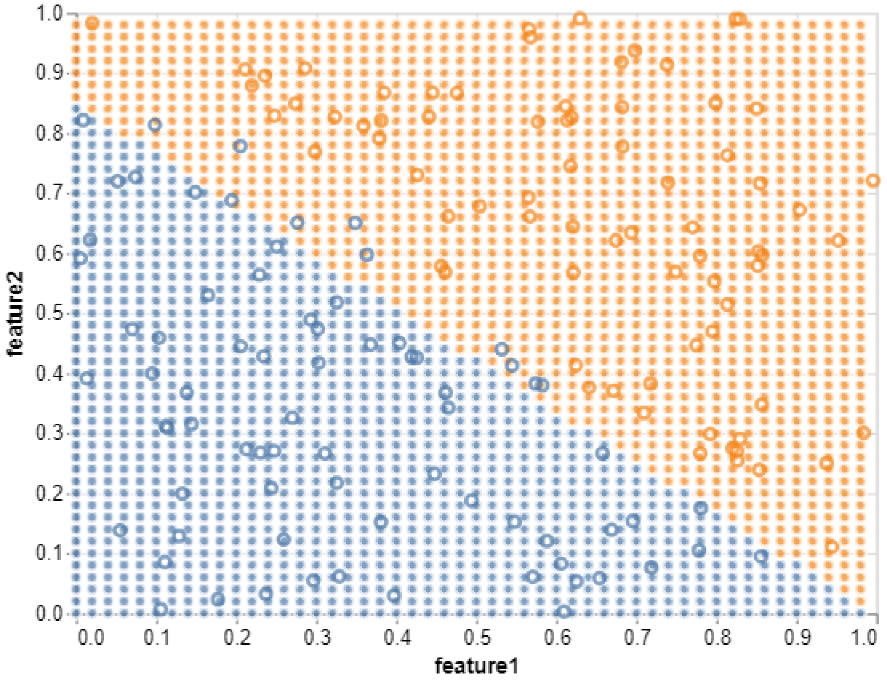


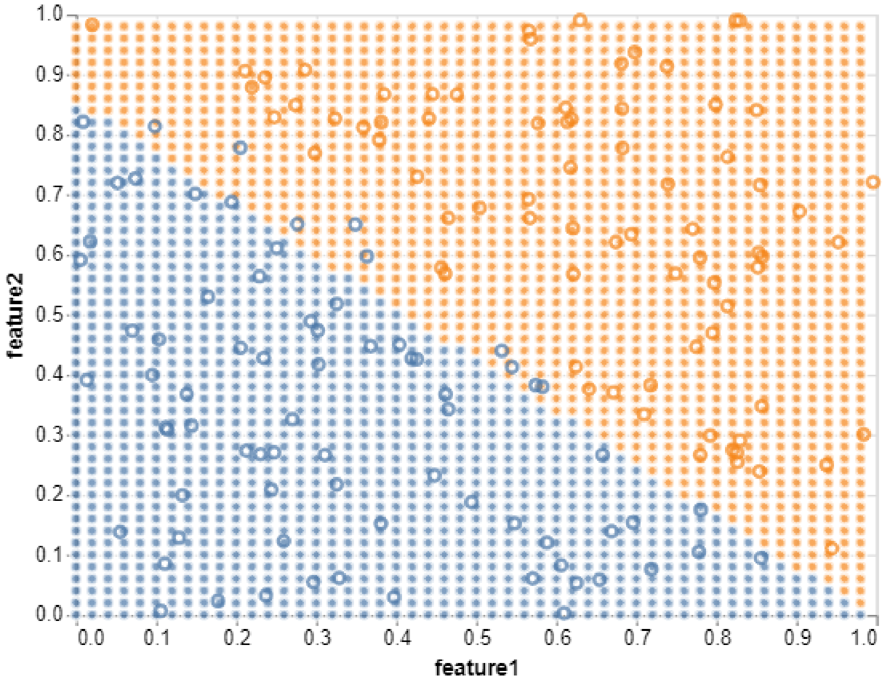
feature2
1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1

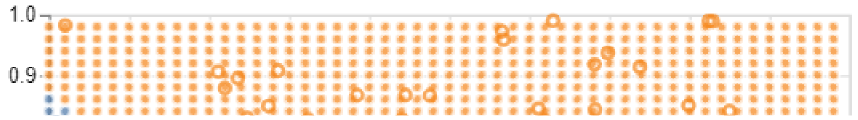


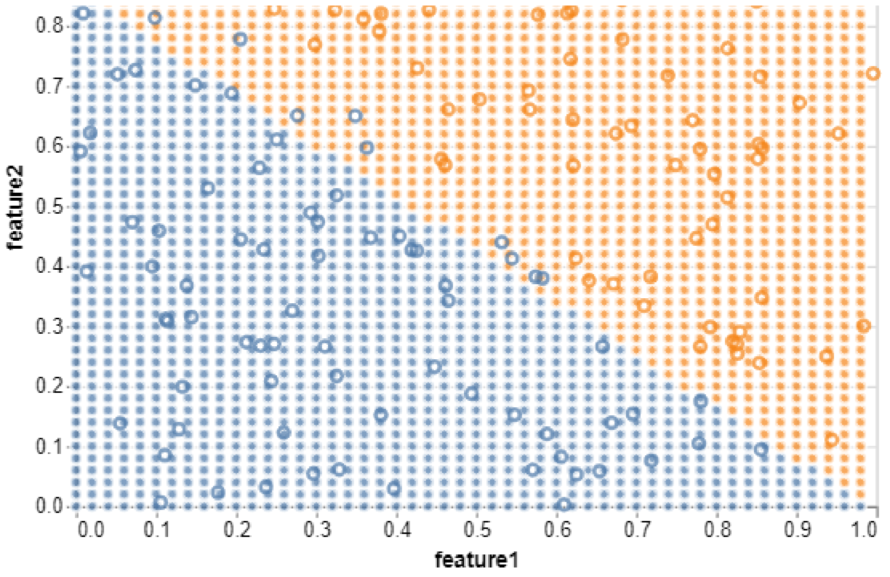
1.0 0.9 0.8 0.7 0.6

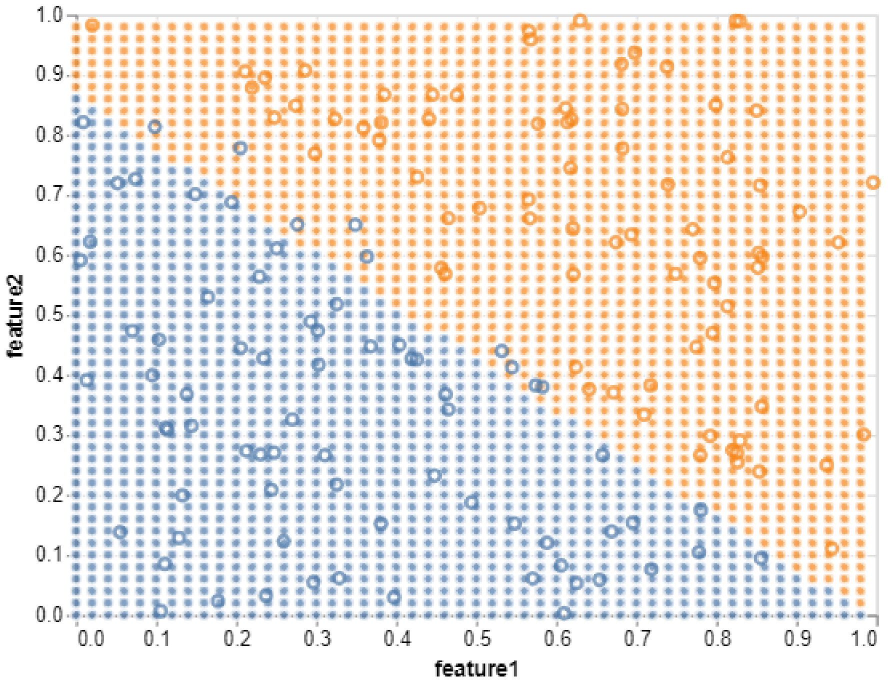

featu
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
feature1

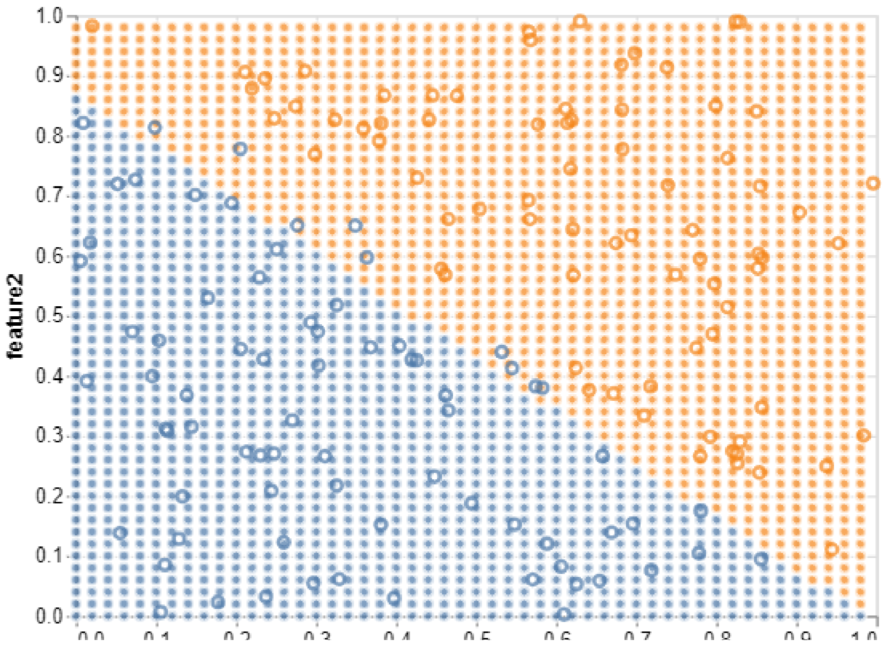

feature2
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
feature1


feature2
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
feature1

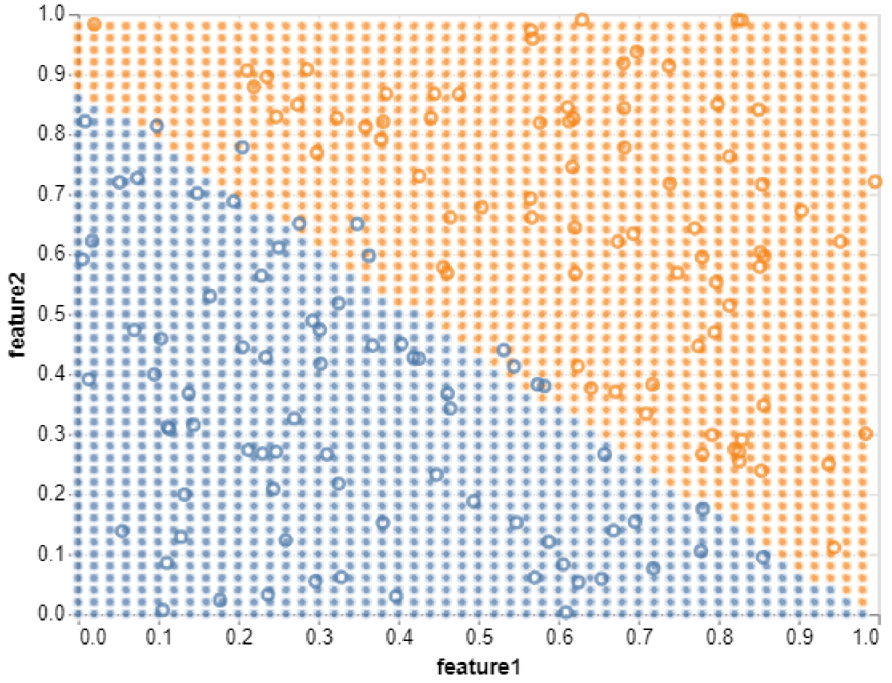

1.0
0.9

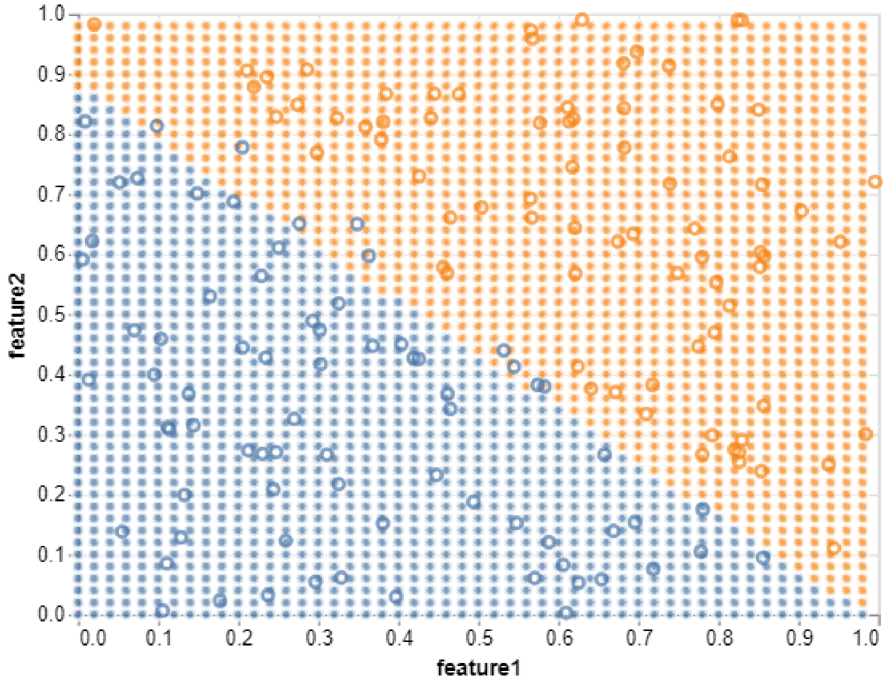

feature2
feature1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

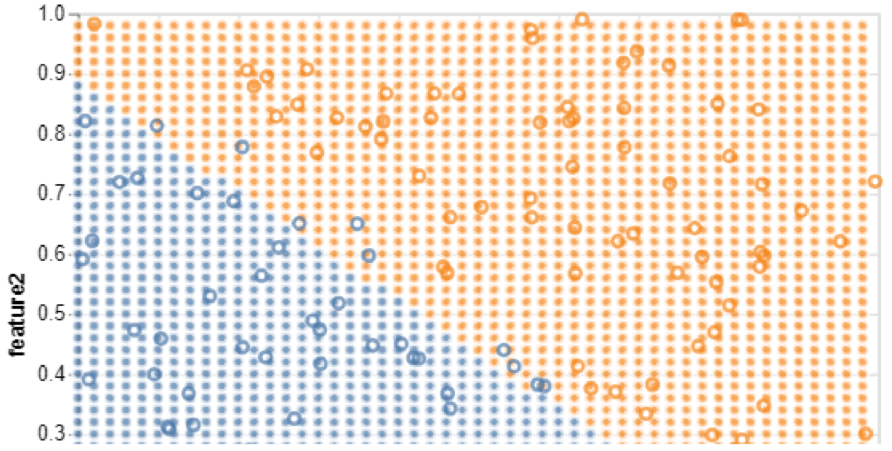

feature2
feature1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

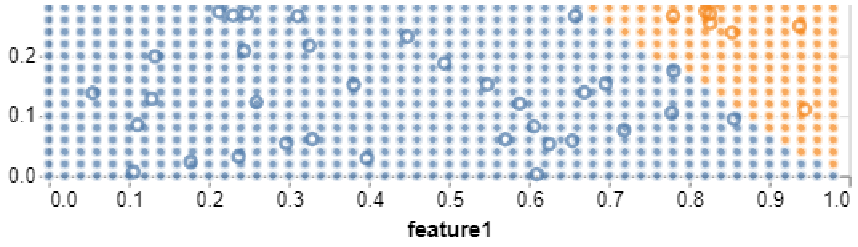

feature2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

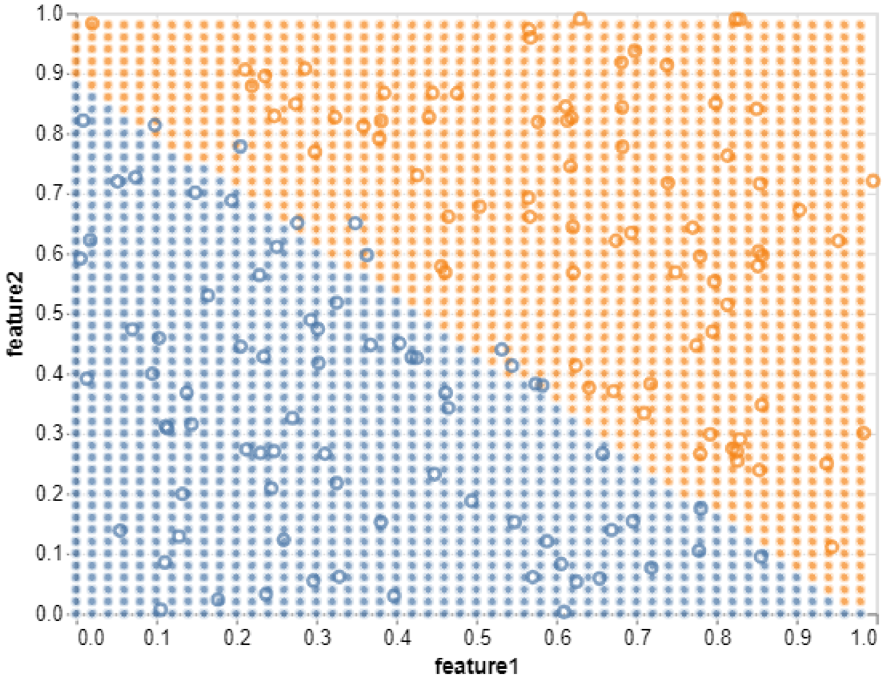
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1

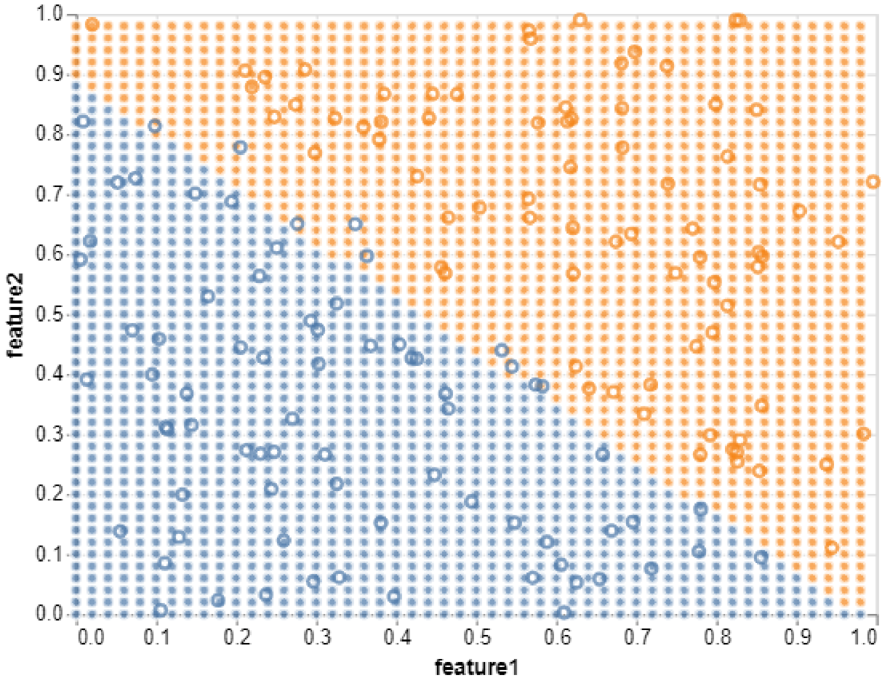


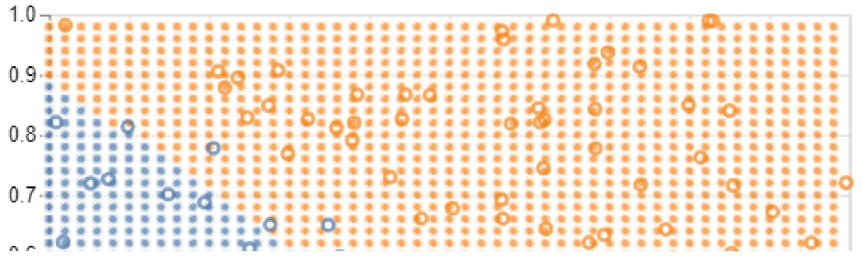


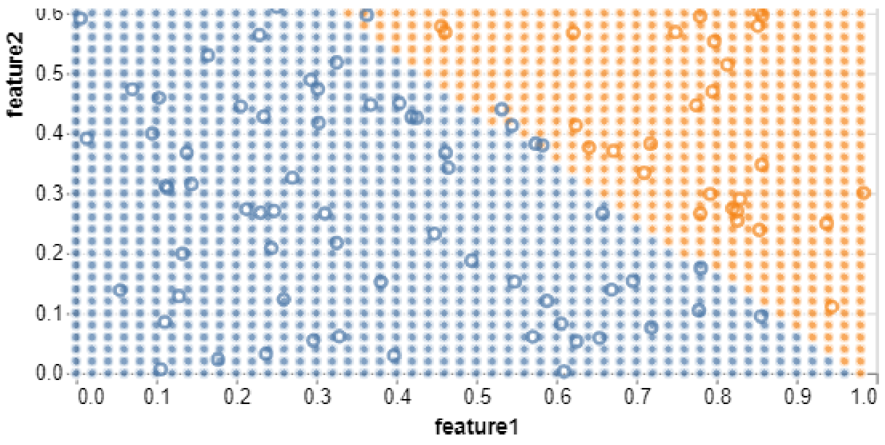


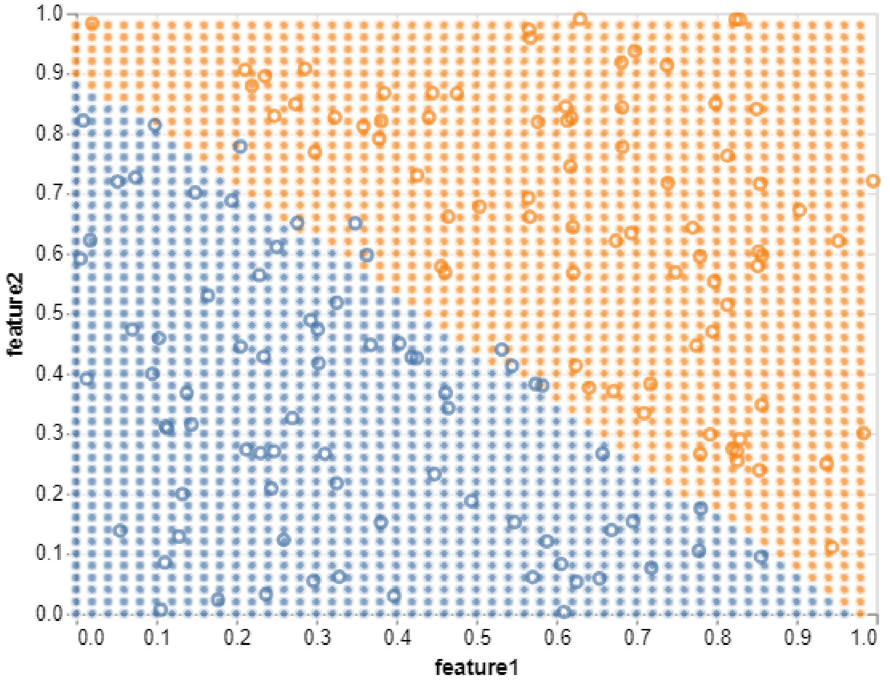

0.2
0.1
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1

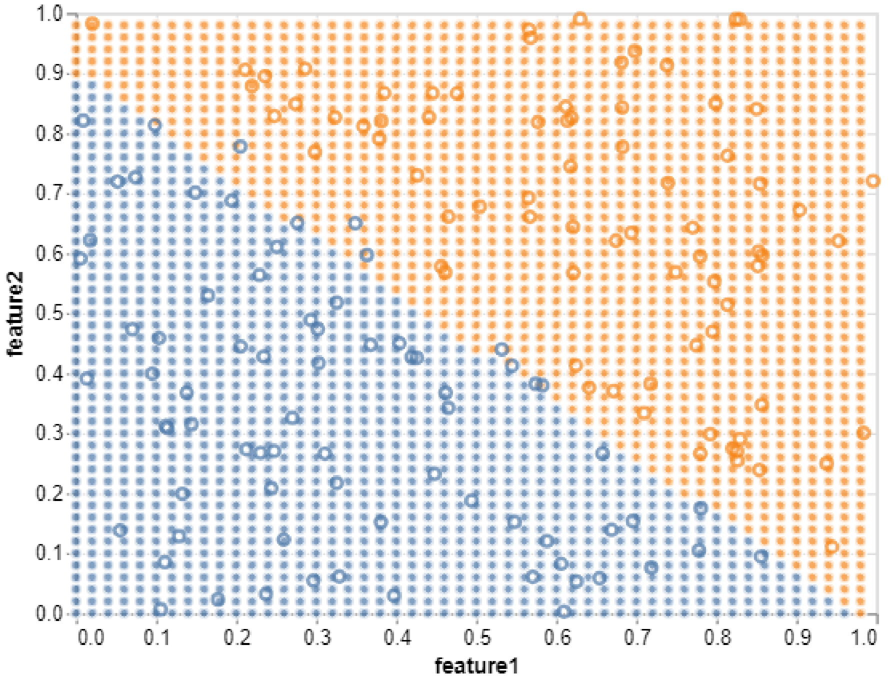

feature2
1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1


feature2
1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
feature1

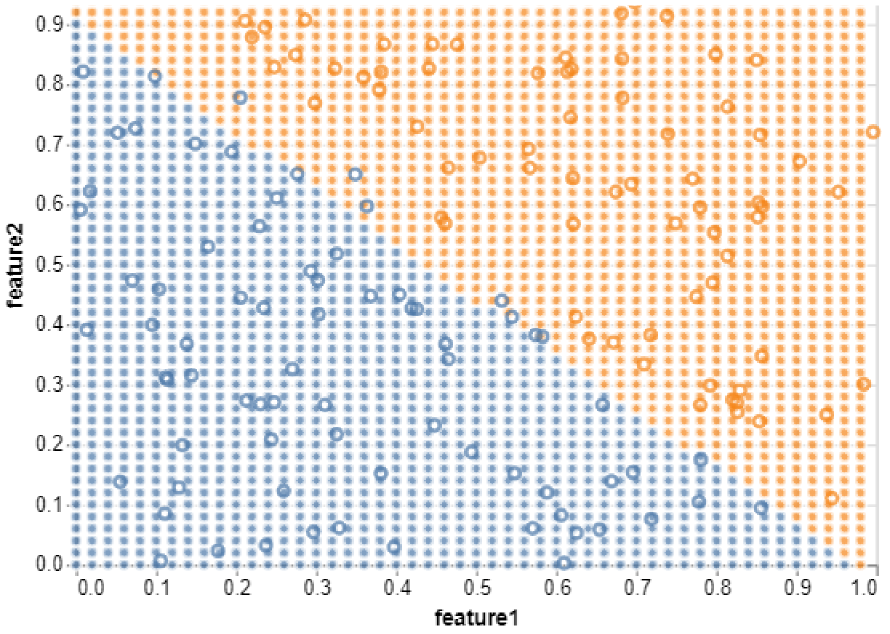

1.0
0.9
0.8
0.7

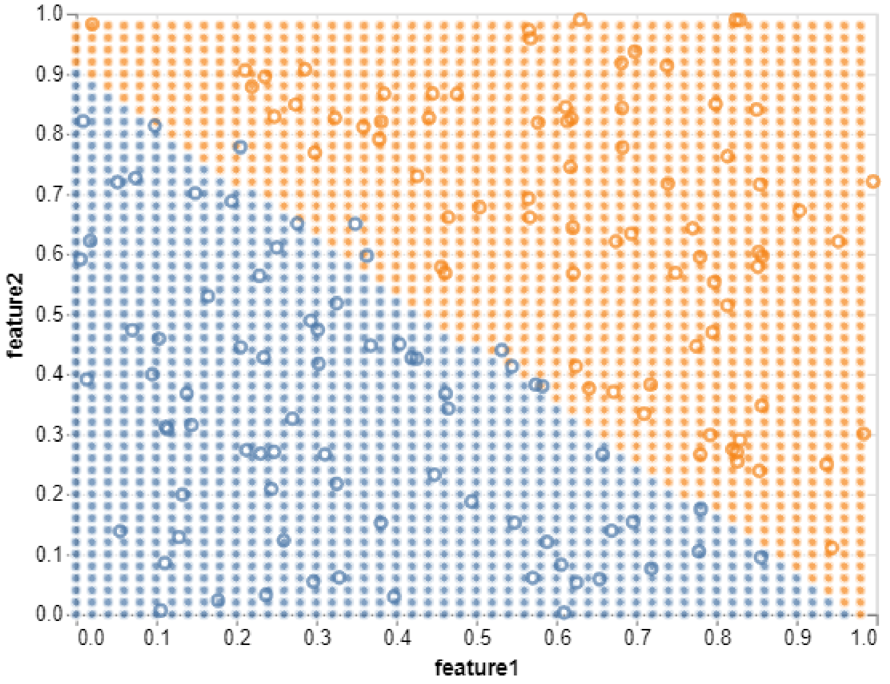

feature2
feature1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

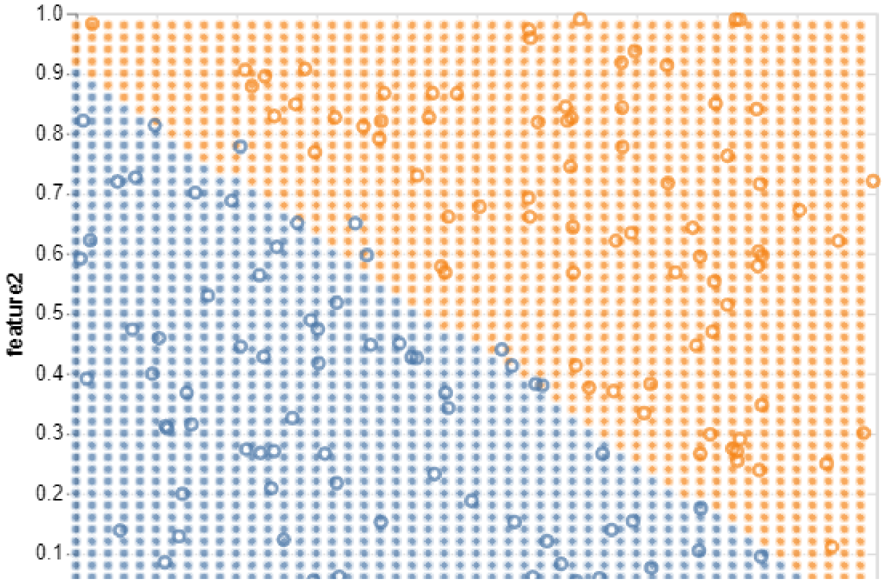

feature2
feature1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

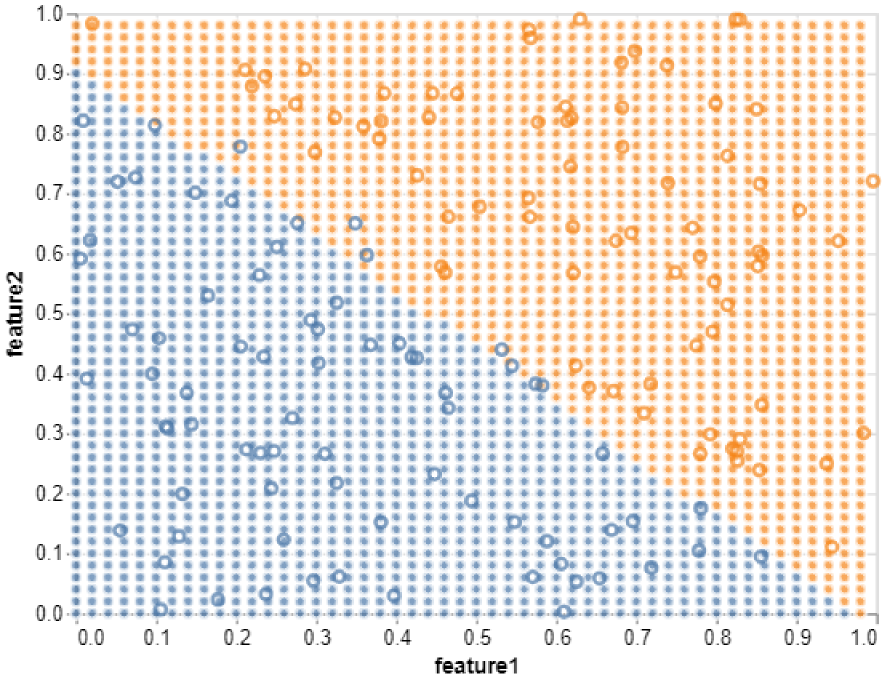

feature2
feature1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

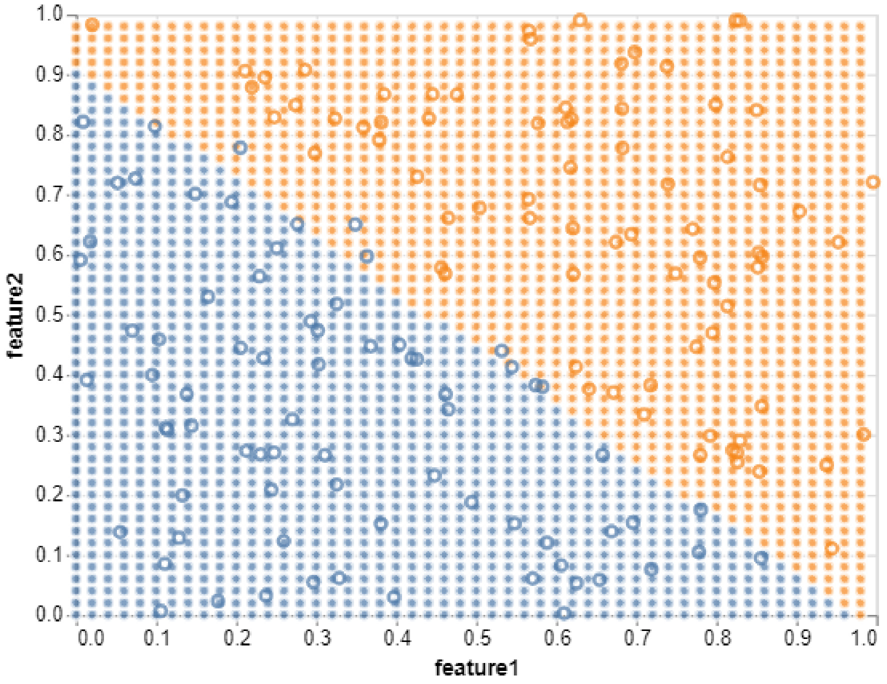
1.0


feature2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
feature1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0


feature2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
feature1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0


feature2
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

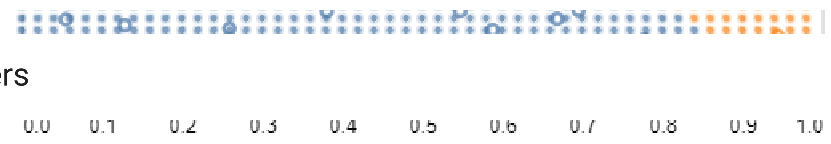




To be more specific, we adjust the parameters opposite the direction of the gradient of the loss with respect to them. Doing so decreases the loss if we are cautious enough to take a small adjustment at a time.

Group Exercise A

Question 0



Icebreakers

Who are other members of your group today?

📝📝📝📝 FILLME

Kera McGovern, Lyra D'Souza, Esther Wang

- What sport would they compete in if they were in the Olympics?

[link text]📝📝📝📝 FILLME

Swimming and Ping pong

- Do they prefer Popcorn or M&Ms?

Popcorn

Double-click (or enter) to edit

📝📝📝📝 FILLME

Question 1

For this question we will use the TensorFlow Playground

[Tensorflow Playground]

- Try changing the learning rate. What happens when it is really low? What happens when it is really high?

```
#📝📝📝📝 FILLME
When the learning rate is low, it continues to go down. When it's high, there are multiple co
```

- On the left hand side there is a noise bar. What happens when you move that? Does this make things easier or harder?

```
#📝📝📝📝 FILLME
It makes things harder.
```

- Can you get some of the harder datasets using different features? Which ones can you get right?

```
#📝📝📝📝 FILLME
Batch size
```

Question 2

Now it's a good time for us to look back at how we created and trained a `KerasClassifier` before. Do you understand what's happening behind the scene now?

```
model = KerasClassifier(build_fn=create_model,
                        epochs=10,
                        rate=1.00,
                        batch_size=20,
                        verbose=False)
df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/simple.csv")
```

Try training the copying the model 3 times with the following settings on train and give the accuracy on test.

1. Learning rate 0.01
2. Epochs 5
3. Batch Size 1

```
#📝📝📝📝 FILLME
def create_model(rate=0.01):
    # Makes it the same for everyone in class
    tf.random.set_seed(2)

    # Create model
    model = Sequential()
    model.add(Dense(1, activation="sigmoid")) # Linear Classifier

    # Compile model
    optimizer = tf.keras.optimizers.SGD(
        learning_rate=rate
    )
    model.compile(loss="binary_crossentropy",
                  optimizer=optimizer,
                  metrics=["accuracy"])
    return model


model = KerasClassifier(build_fn=create_model,
                        epochs=5,
                        rate=0.01,
                        batch_size=1,
                        verbose=False)
```

Which scores the best? Which is the fastest?

Question 3

Run the classifier with `verbose` equal to True. This will print out a lot of additional information

```
model = KerasClassifier(build_fn=create_model,
                        epochs=10,
                        batch_size=20,
                        verbose=True)
model.fit(x=df[["feature1", "feature2"]],
         y=(df["clean"] == "red"))
df["predict_verb"] = model.predict(df[["feature1", "feature2"]])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.7/dist-packages/pandas/core/indexes/base.py in get_loc(self,
key, method, tolerance)
   2897             try:
-> 2898                 return self._engine.get_loc(casted_key)
   2899             except KeyError as err:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in
pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in
```

Describe in words what each part of the output means.

📝📝📝📝 FILLME

```
KevError                                  Traceback (most recent call last)
```

The one with the smallest batch size scored the best. The one with 5 epochs was the fastest!

```
kev  method  tolerance)
```

Unit B

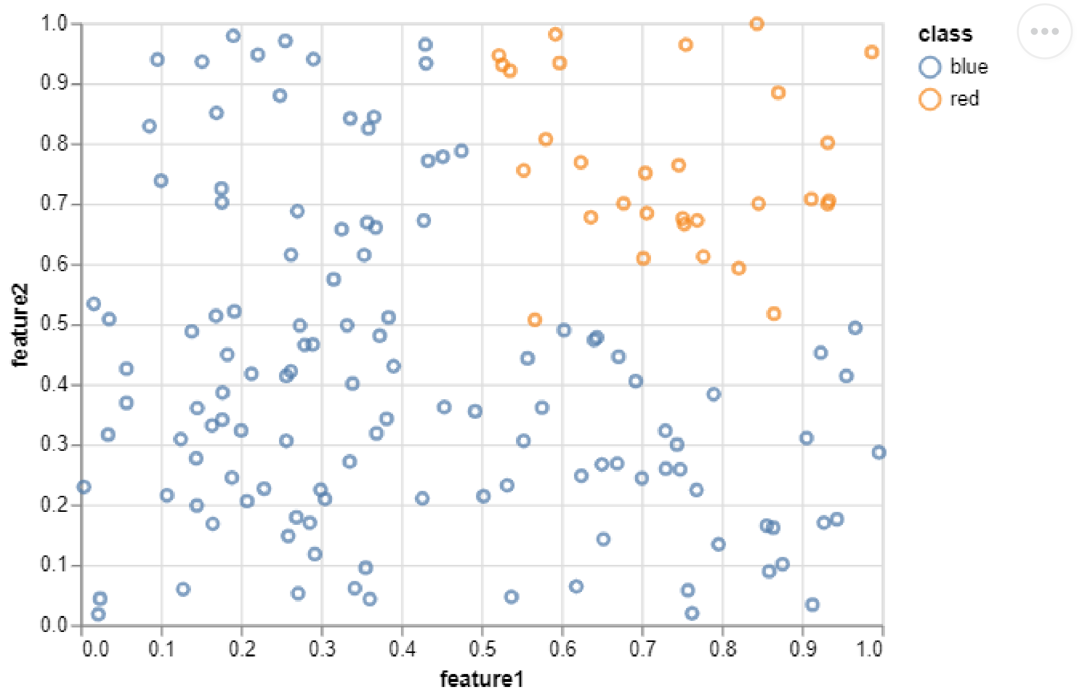
```
   2902             if tolerance is not None:
```

Back to the Playground

In this unit we are going to explore how neural networks can learn to produce some of the features from last class automatically.

Do you remember last class how we had this graph?

```
df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/complex.csv")


chart = (alt.Chart(df)
    .mark_point()
    .encode(
        x = "feature1",
        y = "feature2",
        color = "class"
    ))
chart
```



We saw that this shape was hard for our linear model to learn. Instead we introduced a feature that cut off the top and the bottom. This made it possible for us to learn a linear model.

Let us see how a neural network can handle this problem.

[Example 2]

This model is able to separate out the points without us telling it the features. Because it can do it by itself, we call it a "feature learning" model.

Note that this model has two stages, one that decides on the features and one that decides on the final separation. Because of this we call it a Multi-Layer model.

Let us look back at the playground. If we mouse over the hidden layers, we can see that they each correspond to a linear split. This linear split is how the model decides on the intermediate features.

For instance, do you remember the feature from last week that looked like this? That is exactly the kind of intermediate feature that a model like this would learn.

```
def mkupperfeature(row):
    if row["feature1"] <= 0.5:
        return -1.0
    else:
        return row["feature1"]
```