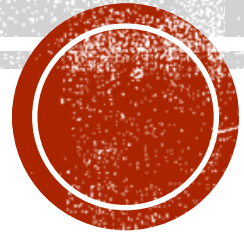
FEATURE ENGINEERING

Dr. Brian Mc Ginley



SKLEARN DATASETS - KNN

- The scales are completely different. Measuring "nearest" counts "mean radius" as much more important than "worst concavity".

```
bc_df = pd.DataFrame(bc.data, columns=bc.feature_names)
print(bc_df[['mean radius', 'texture error', 'worst concavity']].
    describe())
```

```
       mean radius  texture error  worst concavity
count   569.000000     569.000000       569.000000
mean     14.127292       1.216853         0.272188
std       3.524049       0.551648         0.208624
min       6.981000       0.360200         0.000000
25%      11.700000       0.833900         0.114500
50%      13.370000       1.108000         0.226700
75%      15.780000       1.474000         0.382900
max      28.110000       4.885000         1.252000
```

FEATURE SCALING

- At the very least, we can bring the features into a similar range.
- This is a common transformation step at the start of a model: scale each feature separately so they have the same ”size”. That could mean scaling your data so:
  - the smallest value is 0 and the largest is 1.
  - the mean is 0 and standard deviation is 1.
  - something else that makes sense for your data.

FEATURE SCALING

- Sometimes the data we have is not in the best form for the model we are training, so we need to change it. However
  - We don't alter the entire data set
  - This would mean any future data being evaluated by the model would also first have to be transformed
  - We build the transform into the model using make_pipeline

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
model = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(n_neighbors=9)
)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))

0.958041958042
```

- - An improvement on both GaussianNB and the previous KNeighborsClassifier

FEATURE SCALING

- Any data that goes into this pipeline has the first step: the data is transformed, then the transformed data goes into the k nearest neighbour step. We build the transformation into the model. We can chain more things together in a pipeline if we want.
- MinMaxScaler which is also a very commonly used transformation step which makes all the data in the same range, by default the minimum value is changed to 0, the maximum to 1 and everything else scaled accordingly.

TRANSFORM DATA

- Sometimes we want to do a custom transform and build it into the model, for this we need a *FunctionTransformer*

```
from sklearn.preprocessing import FunctionTransformer
def remove_correlation(X):
    X0 = X[:, 0] - X[:, 1]
    X1 = X[:, 1]
    return np.stack((X0, X1), axis=1)

model = make_pipeline(
    FunctionTransformer(remove_correlation),
    LogisticRegression()
)
model.fit(X_train, y_train)
```

- The first step applied remove_correlation to the data, the second step is the training model we want.

FEATURE ENGINEERING

- We have seen:
  - Just use the original features.
  - Calculate powers of the original features (*Polynomial_Features*).
  - Scale features to tidy their min/max (*MinMaxScaler*) or mean/?stddev (*StandardScaler*).
  - Calculate anything else we want/need (*FunctionTransformer*).

FEATURE ENGINEERING

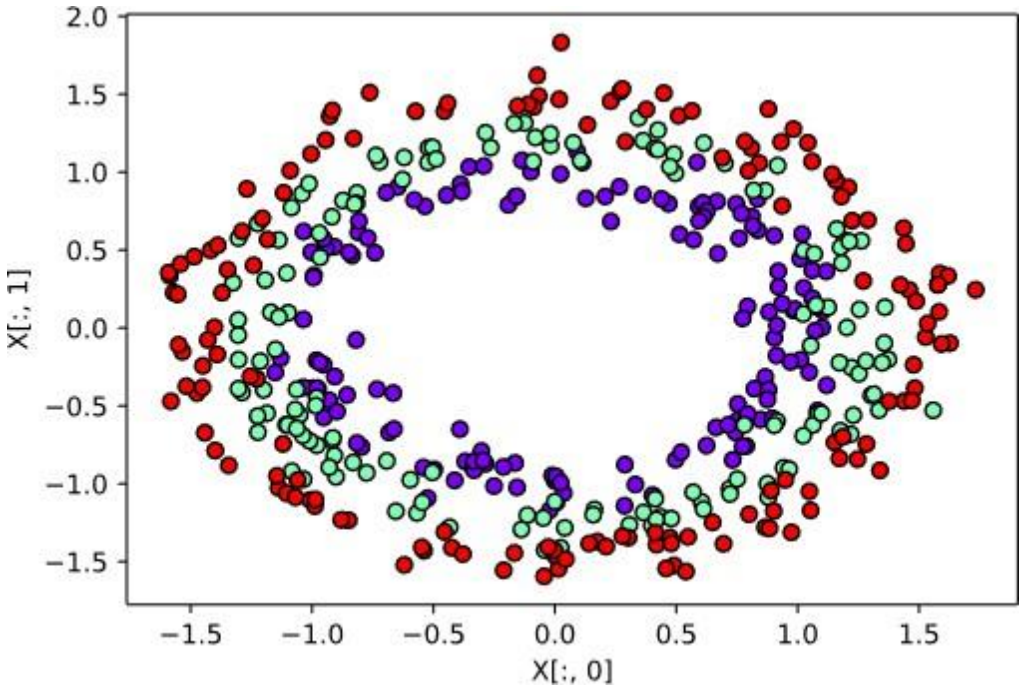
- Not surprisingly, the kind of X values we have for each training point have a huge effect on what we can learn/predict.
- The components of each X value are the *features*.
- **A combination of understanding our data and the model capabilities/training can help us be much more successful.**

FEATURE ENGINEERING

- We can design whatever features we think will give the model something to work with. As an example:
- With that data and a kNN model:

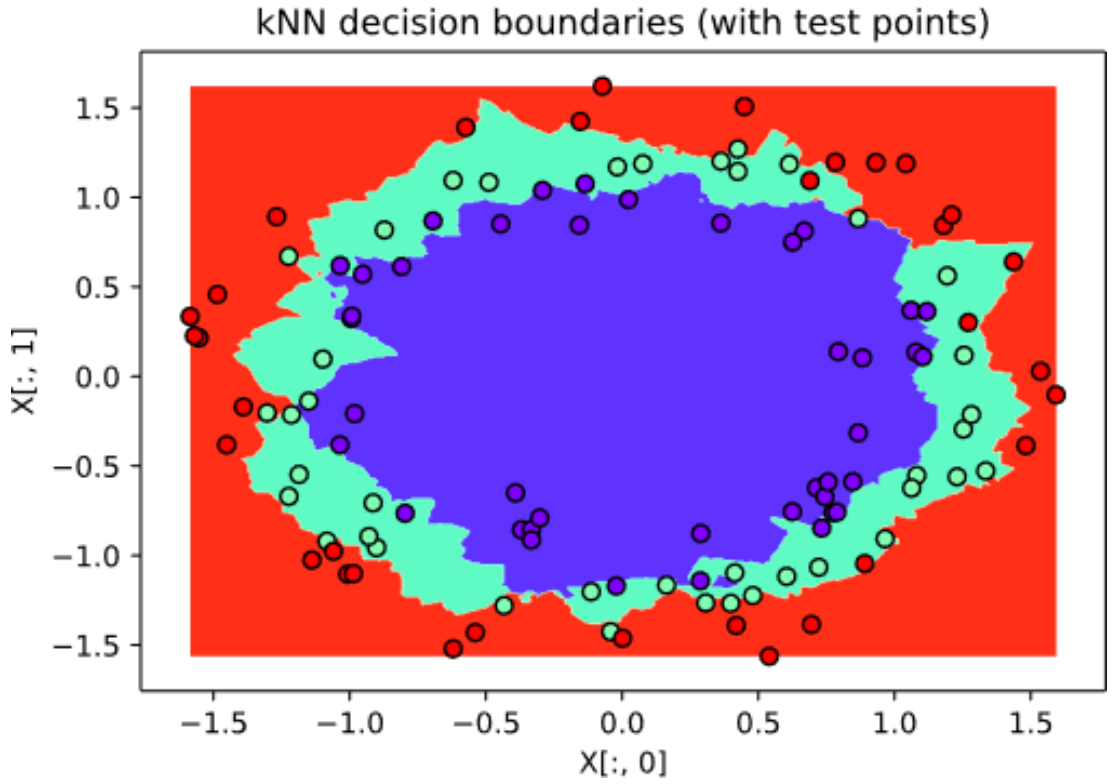
```
model = KNeighborsClassifier(7)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))

0.805309734513
```



FEATURE ENGINEERING

- It doesn't have enough training data to really capture what's going on, and overfits the training data.



FEATURE ENGINEERING

- Maybe if we manually created an extra feature of how far away the point is from (0, 0), it would improve the results.
  - Add a transformer:

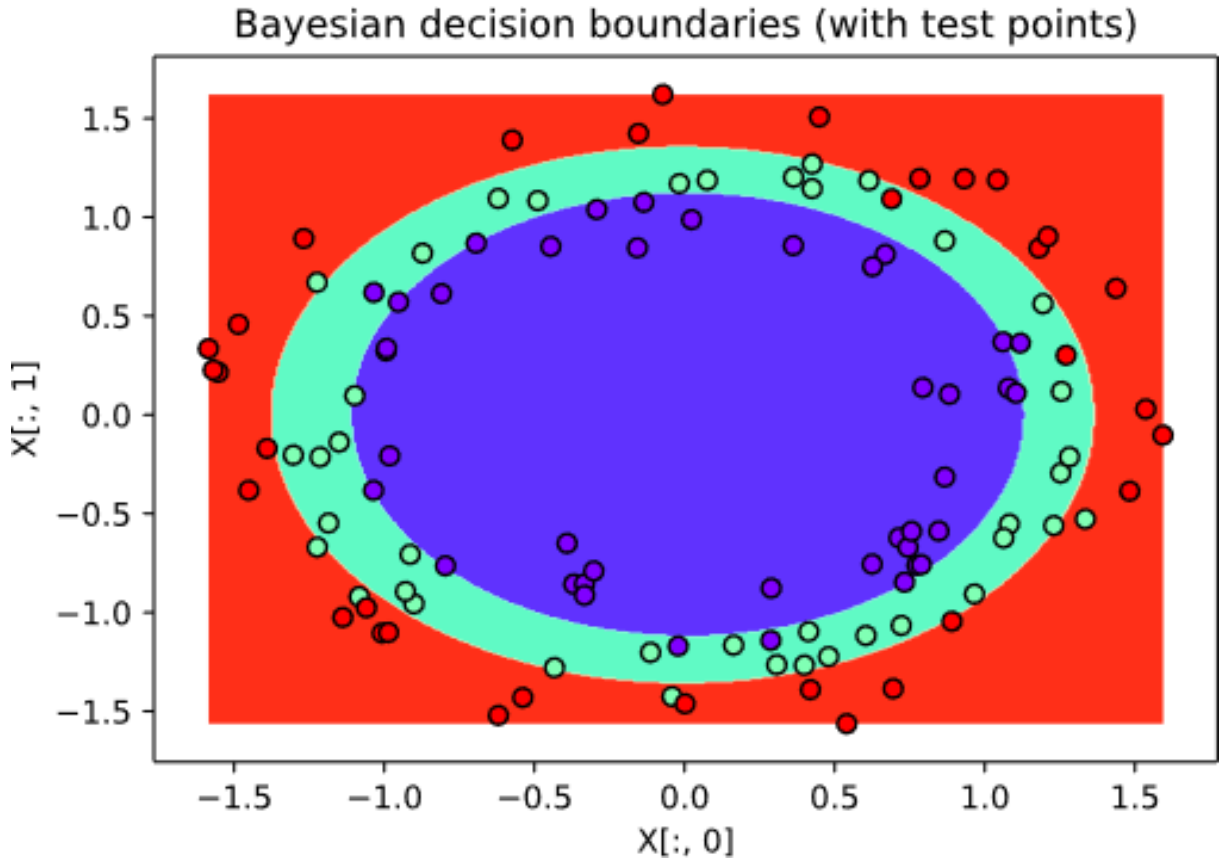
```
def add_radius(X):
    X0 = X[:, 0]
    X1 = X[:, 1]
    R = np.linalg.norm(X, axis=1) # Euclidean norm
    return np.stack((X0, X1, R), axis=1)

model = make_pipeline(
    FunctionTransformer(add_radius),
    GaussianNB())
model.fit(X_train, y_train)
print(model.score(X_test, y_test))

0.893805309735
```

FEATURE ENGINEERING

- The model does better with the right features



FEATURE ENGINEERING

- Other problems that may arise is having
  - More features than train & test data.
  - Too large of a data set (processor/memory).
- Sometimes we have to group common features together
  - Example:
    - We observe multiple weather values (rainfall, temperature) every day. This gives us 365 points (times the number of weather features) for one year. It may make sense to average these values for each month and cut down the number of points to 12 for a year (times the number of weather features). This comes from knowing the data.
    - We could also average over each day in a month, 1st of month, 2nd of month etc. so down to 31 points….however this would make no sense for grouping together – you need to know your data.

HIGH-DIMENSIONAL DATA KNN (SOURCE WIKIPEDIA)

- For high-dimensional data (e.g., with number of dimensions more than 10) dimension reduction is usually performed prior to applying the k-NN algorithm in order to avoid the effects of the curse of dimensionality.
- The curse of dimensionality in the k-NN context basically means that Euclidean distance is unhelpful in high dimensions because all vectors are almost equidistant to the search query vector.
- Feature extraction and dimension reduction can be combined in one step using principal component analysis (PCA), linear discriminant analysis (LDA), or canonical correlation analysis (CCA) techniques as a pre-processing step, followed by clustering by k-NN on feature vectors in reduced-dimension space. This process is also called low-dimensional embedding.
- For very-high-dimensional datasets (e.g. when performing a similarity search on live video streams, DNA data or high-dimensional time series) running a fast approximate k-NN search using locality sensitive hashing, ”random projections”, ”sketches” or other high-dimensional similarity search techniques from the VLDB toolbox might be the only feasible option.