ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python Program

CHAPTER 7: CLASSIFICATION MODELS

Chapter Objectives

In this chapter, we will:

- Understand the use cases for Classification models
- Discuss and compare various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Classification

- There are many business use cases for wanting to predict whether a record will fall into one category or another
  - Is a credit card swipe fraudulent or not?
  - Does a patient have a disease?
  - Will an applicant be a profitable customer?
- Classification can make such predictions by using historical data to train the model to look for patterns
- To see how good a job the model does, you test it with another set of data that was not used to train the model
  - By comparing the known values to the predicted ones, you can judge how well the model performs
- If the model guesses better than a coin flip or random guess, it may not be perfect, but it's better than nothing and could be used to create actionable business decisions with a best guess
- There are many different algorithms that can do classification, so it's best to run the same data through many different models to see which works best for your data

Steps to Classification

- There are some consistent steps you need to do regardless of which algorithm you use, although some models have different requirements
- Usually categorical data needs to be re-encoded as either a series of numbers or as dummy encoded data
  - There's really no consistency among different algorithms
  - Learn from experience and examples which algorithms require which style of data
- All models should take the data set and split it into a training and testing set
- First, you fit the model with the training set
  - Can take some time for large datasets
- Once the model is trained, you can see how good it is at making predictions by using a predict function and comparing those values to known values for the variable you are trying to predict
- After you've picked the model that does the best job, you can use it to predict values either individually or in a batch for new data as it comes in

Notes on Classification

- Often you are trying to predict an either/or value
  - Is a card swipe fraudulent or legitimate?
  - Does a patient have cancer or not?
- Not limited to just two choices, you could predict whether a record falls into a category with many different values
  - It just gets trickier sometimes to interpret the results
- The math and techniques behind the scenes can get complicated, but you don't really need to know any of it to use the algorithms
- Just knowing how to identify that you need a classification model and how to prep the data and interpret the results is often good enough to get started
- As you get more sophisticated, you can learn the math behind the scenes to tweak the results and try to get better results

Applying Classification

- Let's look at some data first, and use this with several different algorithms

```
import pandas as pd
df = pd.read_csv('CreditCardFraud.csv')
print(df.shape, df.columns)
print(df.isFraud.value_counts())
print(df.type.value_counts())

(6362620, 11) Index(['step', 'type', 'amount', 'nameOrig',
'oldbalanceOrg', 'newbalanceOrig',       'nameDest', 'oldbalanceDest',
'newbalanceDest', 'isFraud',       'isFlaggedFraud'],
dtype='object')

0    63544071       8213
Name: isFraud, dtype: int64
CASH_OUT    2237500
PAYMENT     2151495
CASH_IN     1399284
TRANSFER     532909
DEBIT         41432
Name: type, dtype: int64
```

Change Categorical Column

- Change the type categorical column to codes and keep certain columns

```
columns = ['type', 'amount', 'oldbalanceOrg', 'newbalanceOrig',
           'oldbalanceDest', 'newbalanceDest', 'isFlaggedFraud',
           'isFraud']
df = df[columns]
df.type = pd.Categorical(df.type).codes
```

- Split the data into training and testing sets
  - Check the ratios of the two sets are about the same

```
from sklearn.model_selection import train_test_split
from sklearn import preprocessing as pp
trainX, testX, trainY, testY = train_test_split \
        (df[df.columns[:-1]], df.isFraud, \
        train_size = train_size, test_size = test_size)
print(testY.value_counts())
print(trainY.value_counts()/trainY.count())
print(testY.value_counts()/testY.count())
print(trainX.head(10))
```

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Naive Bayes

- Based on Bayes theorem and involves calculating lots of probabilities of events occurring based on prior observations of conditions related to that event
- Pros
  - It's easy and fast to train often outperforming more complex algorithms
  - Performs well with categorical data
  - Works well for multi-class predictions
- Cons
  - Doesn't do well if a category is found in the test set that is missing from the training set
  - Not useful for tweaking the false negatives (FN)/false positives (FP)

Apply Naive Bayes

- Load the module, create a model and train it

```
from sklearn.naive_bayes import GaussianNB
modelNB = GaussianNB()
modelNB.fit(trainX, trainY)
```

- Run a prediction on the reserved testing set and compare the predicted values to the known values

```
predY = modelNB.predict(testX)

# evaluate_predictions is a helper function we wrote
evaluate_predictions(testY, predY)
```

Interpret the Results

- The confusion matrix shows/compares the predictions to the known values
  - 1,263,180 were predicted to be good charges and were indeed good
  - 285 were predicted to be fraudulent and were indeed fraudulent
  - 7667 were predicted to be good but were in fact fraudulent (false positive or Type I error)
    - 1,263,180 + 7667 = 1,270,847
  - 1392 were predicted to be fraudulent but were in fact good (false negative or Type II error)
    - 1392 + 285 = 1677

```
[[1263180    7667]
[   1392     285]]

[[99.28810773  0.60250337]
 [ 0.1093889   0.71189227]]

0   1270847
1   1677

1272524
```

- Results as percentages
  - 99.28% correct
  - .71% incorrect
  - .60% false positive
  - .10% false negative
- Overall pretty good at predicting

Save and Load Model

- To save the results of a lengthy training process
  - `pip install joblib`

```
from joblib import dump, load
dump(modelNB, 'modelNB.joblib')
```

- To reload a saved model

```
modelNB2 = load('modelNB.joblib')
predY = modelNB2.predict(testX)
evaluate_predictions(testY, predY)
```

Decision Trees

- Data is split up based on some factor among the independent variables, then evaluated for how good a job it did
- Recursively keeps applying this algorithm over and over until it comes up with a good set of rules
- Pros
  - It's easy
  - Performs well with categorical data and continuous data
  - Transparency lets you see how it made its choices
- Cons
  - Calculations take a lot longer as you add more and more columns
  - Becomes difficult to understand the decision tree as it gets larger

Apply Decision Tree

- Load the module, create a model and train it

```
from sklearn.tree import DecisionTreeClassifier
modelDT = DecisionTreeClassifier()
modelDT.fit(trainX, trainY)
```

- Run a prediction on the reserved testing set and compare the predicted values to the known values

```
def important_features(model, columns):
   return pd.DataFrame(model.feature_importances_, \
         columns=['Importance'], \
         index = columns).sort_values(['Importance'], \
         ascending = False)

predY = modelDT.predict(testX)
evaluate_predictions(testY, predY)
print (important_features(modelDT, trainX.columns))
```

Interpret the Results

- Results of the confusion matrix are interpreted exactly as before
  - The numbers different so see which did a better job
- Decision Trees have additional information about what factors contributed to its decisions
  - Calculated by number of samples that reach the node divided by the total number of samples
  - The higher the number, the more important the feature

	Importance
oldbalanceOrg	0.432194
newbalanceDest	0.228966
amount	0.161474
oldbalanceDest	0.087661
newbalanceOrig	0.069822
type	0.019578
isFlaggedFraud	0.000305

- Based on these results, we can see the most important factors as to how it decided a transaction was legitimate or fraudulent where `oldbalanceOrg` and `newbalanceDest` is followed by the amount
  - The remaining columns were of less importance

Random Forest

- Creates Decision Trees on randomly selected samples of the training set
- Performs multiple iterations and gets prediction results
- Votes on the best random sample
- Pros
  - Often highly accurate due to the strength of multiple predictions
  - Usually does not suffer from overfitting
  - Can see the relative feature importance which is useful in revising the model
- Cons
  - Slow to generate because of multiple iterations
  - Compared to a Decision Tree you cannot really see the path of the tree

Apply Random Forest

- Load the module, create a model and train it

```
from sklearn.ensemble import RandomForestClassifier
modelRF = RandomForestClassifier(n_estimators=10)
modelRF.fit(trainX, trainY)
```

- Run a prediction on the reserved testing set and compare the predicted values to the known values

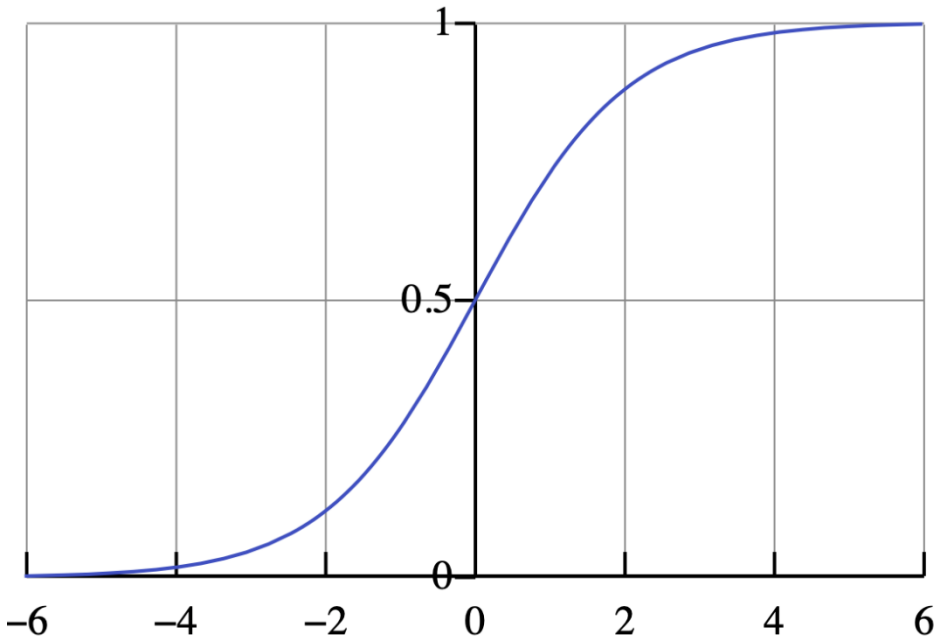
```
predY = modelRF.predict(testX)
evaluate_predictions(testY, predY)

import pandas as pd
feature_imp = pd.Series(modelRF.feature_importances_,\
    index=trainX.columns).sort_values(ascending=False)
print (feature_imp)
```

Logistic Regression

- Yet another alternative to categorizing, but with a twist
  - Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- The math behind it involves logarithms and finding coefficients of the independent variables

$$\ell = \log \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$



Logistic Regression (continued)

- Data needs to be dummy encoded skipping one value as a reference value
- Pros
  - Works better in cases with low signal to noise ratio
  - Allows for tweaking of false positives and false negatives
  - Transparency lets you see how it made its choices
- Cons
  - Does not perform well with too many features (independent variables)
  - Not good with large number of categorical values within a feature because of dummy encoding

Apply Logistic Regression

- Need to dummy encode categorical features
  - Use this helper function to make that easier

```
from sklearn.model_selection
import train_test_splitfrom sklearn
import preprocessing as pp
def dummy_code(data, columns, drop_first = True):
   for c in columns:
      dummies = pd.get_dummies(data[c], prefix = c, \
      drop_first = drop_first)
      i = list(data.columns).index(c)
      data = pd.concat([data.iloc[:,:i], dummies, \
             data.iloc[:,i+1:]], axis = 1)
return data

df2 = dummy_code(df, ['type'], drop_first = True)
trainX, testX, trainY, testY = train_test_split(df2.iloc[:,
df2.columns != 'isFraud'], df2.isFraud, train_size = train_size,
test_size = test_size)
print (testX.columns)
print (testX.head())
```

Apply Logistic Regression (continued)

- Create and train the logistic model

```
from sklearn.linear_model import LogisticRegression
modelLR = LogisticRegression(multi_class='auto', solver='lbfgs')
modelLR.fit(trainX, trainY)
print (modelLR.coef_)

[[-2.34708708e-08 -2.08083063e-09 -3.31118886e-07
  -8.31500683e-10  -8.26614837e-04  8.30307380e-04
  -9.09162361e-04  7.04021006e-07  -2.41542762e-06
  1.05305835e-11]]
```

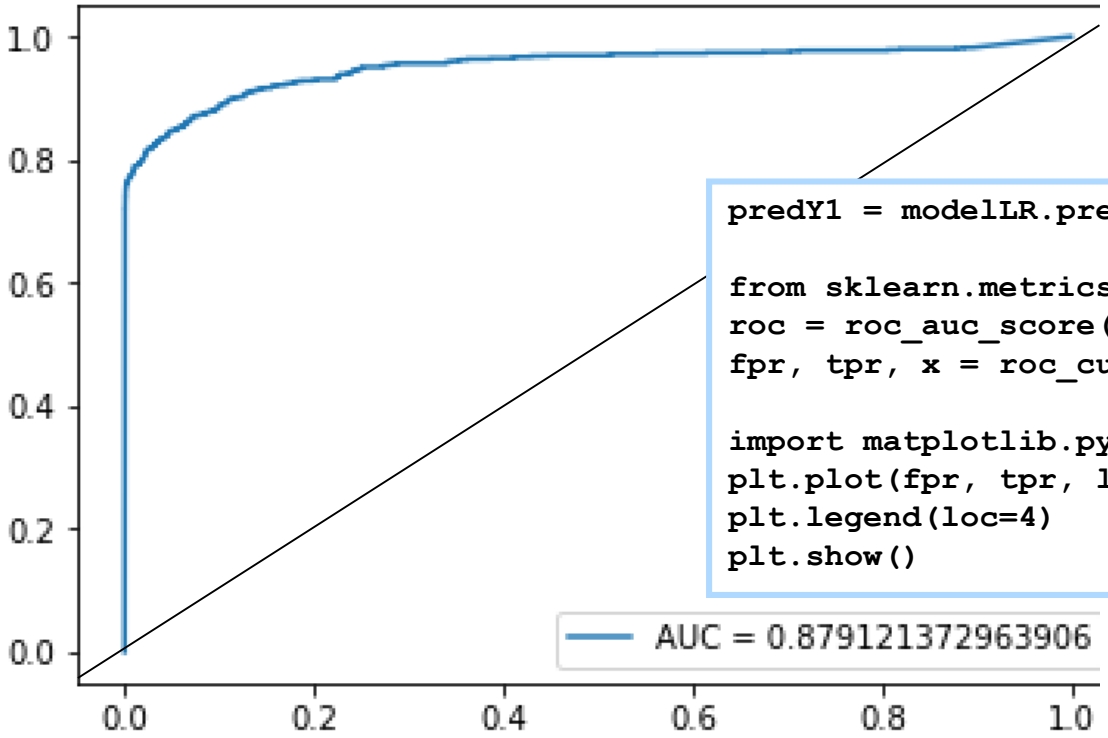
Interpret the Results

- The coefficients show how the math formula is trained
  - Multiply each feature by its coefficient and sum them up to get the probability
  - ln(p/(1-p)) = β + β1X1 + β2X2 + β3X3 + … + βnXn
- Confusion matrix is interpreted just like all the other models

```
0.998001609399901 3 0.0019983906000987013
[[1268705    2162]
 [    381    1276]]
PC FP
FN PW
[[9.98001609e+01 1.69898564e-01]
 [2.99404962e-02 1.99839060e-01]]
```

ROC Curve

- Shows the tradeoff between accuracy and sensitivity in adjusting the False Positives
- The closer the curve is to the left or top border, the more accurate it is
- The closer to the 45 degree line, the less accurate it is




```
predY1 = modelLR.predict_proba(testX)

from sklearn.metrics import roc_auc_score, roc_curve
roc = roc_auc_score(testY, predY)
fpr, tpr, x = roc_curve(testY, predY1[:,1])

import matplotlib.pyplot as plt
plt.plot(fpr, tpr, label = 'AUC = ' + str(roc))
plt.legend(loc=4)
plt.show()
```

Different Thresholds

- Additionally, you can do the prediction probabilities
  - Set a threshold probability to determine whether the prediction is positive or negative
  - Allows you to tweak the accuracy, false positives and false negatives
- Use the `predict_proba` function instead

```
predY = modelLR.predict_proba(testX)
print (predY[:10])

[9.99999999e-001 1.72171774e-113]    # 99% likely positive
[8.26513185e-001 1.73486815e-001]    # 82% likely positive
[6.28732003e-178 1.00000000e+000]    # 100% likely negative
[5.20000000e-001 4.80000000e-001]    # 52% likely positive
```

Tweak the Results

- Run the prediction on the same trained model several times with different probability thresholds and compare the accuracy and FP/FN

```
predY = modelLR.predict_proba(testX)
print (predY[:10])
print (modelLR.score(testX, testY))
for threshold in range(30, 91, 10):
   predY1 = np.where(predY[:,1] >= threshold/100, 1, 0)
   mse = np.mean((predY1 - testY)**2)
   evaluate_predictions(testY, predY1)
```

- Highest accuracy turned out to be about 70 in this case
- Lower threshold yielded more false positives, fewer false negatives
  - Tune the threshold to fit the business case of whether you favor FP or FN

Neural Networks

- Simulates the way the human brain solves
  - Uses a perceptron, algorithm, or function run in multiple layers
- Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- Pros
  - Often perform better than others which can be important where accuracy is desired (predicting cancer)
  - Good for unusual data like image, video, audio
- Cons
  - Black box, you don't know how it made its decision
  - Not appropriate in cases where transparency is important
  - Require a lot more data to train than other models
  - Computationally expensive
- Cool visualization of Neural Network from Google
  - https://playground.tensorflow.org/

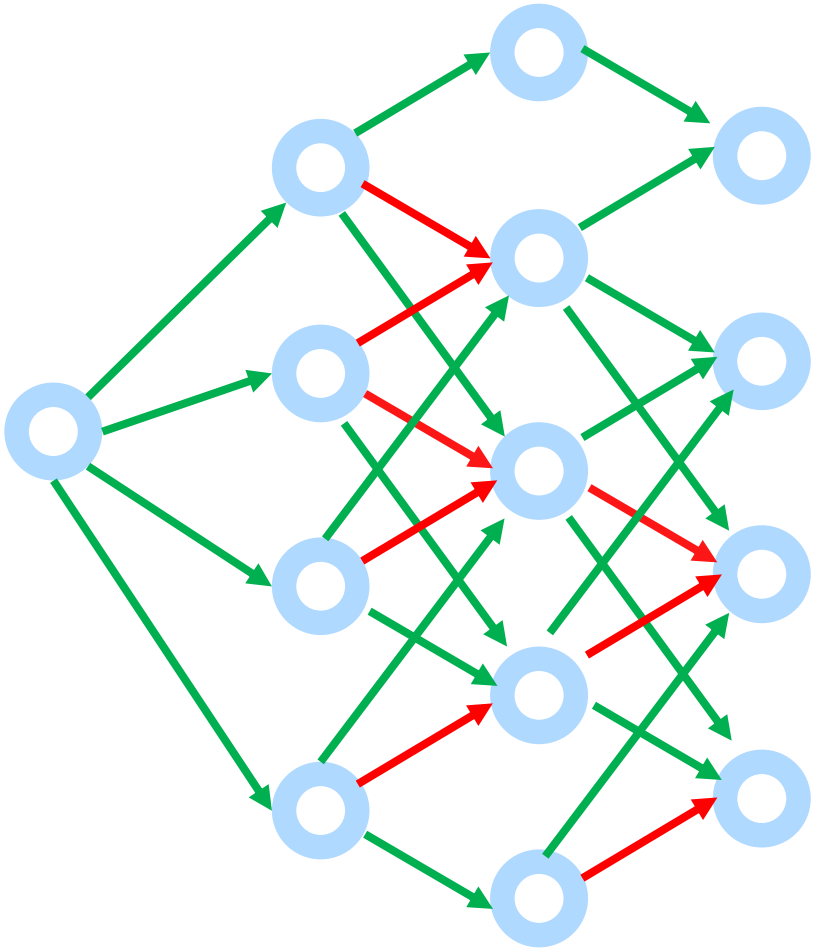
Neural Network Visualized

Data

Input Layer

Hidden Layer

Output Layer



Apply Neural Network

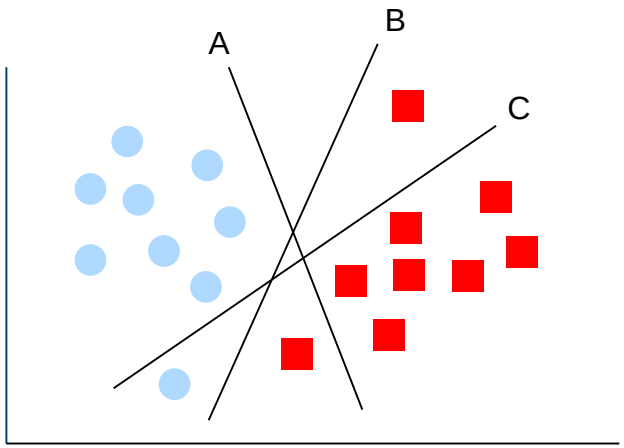
- Need to dummy encode categorical features
  - This time keep the first element
- Also works best if you rescale the numeric values
  - Rescale the whole dataset before splitting it or else the results differ

```
from sklearn.model_selection import train_test_split
from sklearn import preprocessing as pp
# rescale the data
df2 = dummy_code(df, ['type'], drop_first = False)
print (df2.columns)
df2[['amount',  'oldbalanceOrg', 'newbalanceOrig',
'oldbalanceDest', 'newbalanceDest']] /= df2[['amount',
'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest',
'newbalanceDest']].max()
trainX, testX, trainY, testY =
train_test_split(df2.iloc[:,df2.columns != 'isFraud'],
df2.isFraud, train_size = train_size, test_size = test_size)
```

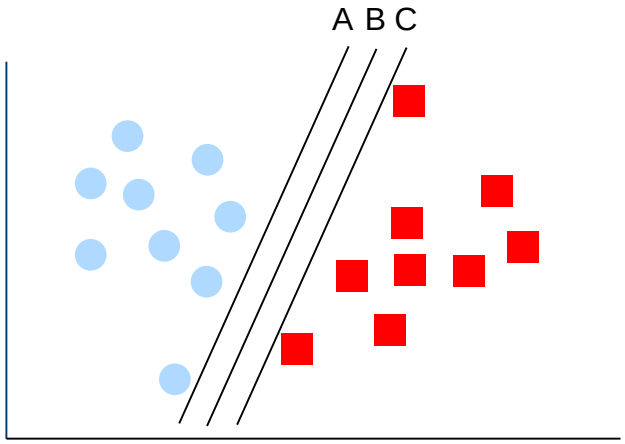
- Look at the results just like all the other classification models

Support Vector Machine

- Support Vector Machine attempts to find the hyperplane that separates the two classes better—in this case, B clearly does that

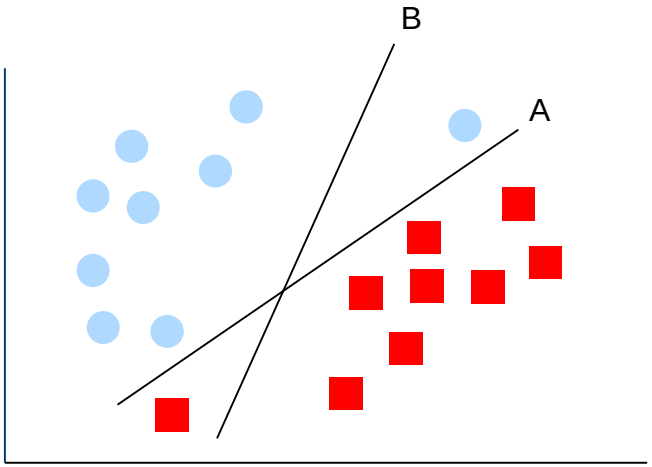


- Next, it tries to find the right hyperplane that maximizes the distances from that plane to the nearest data points in either group—again, it is B in this case
  - This is called a margin

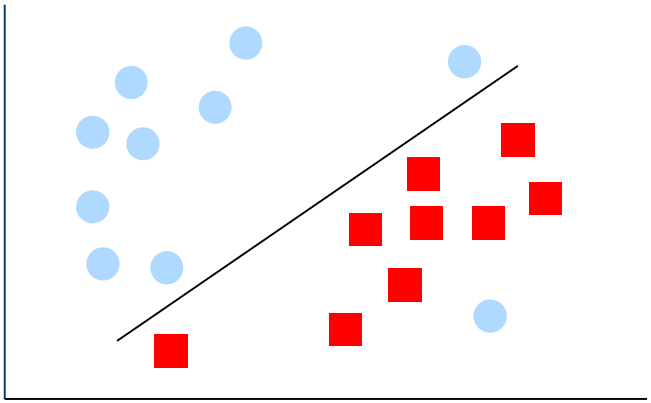


Support Vector Machine (continued)

- In this case, B has a bigger margin but A is the plane it determined did a better job of classifying them so it would choose A

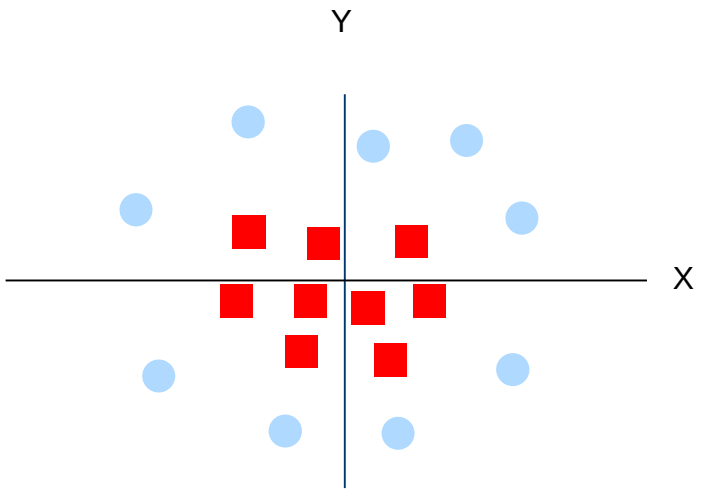


- Here we have a blue outlier, and SVM has a feature that allows such outliers to be ignored

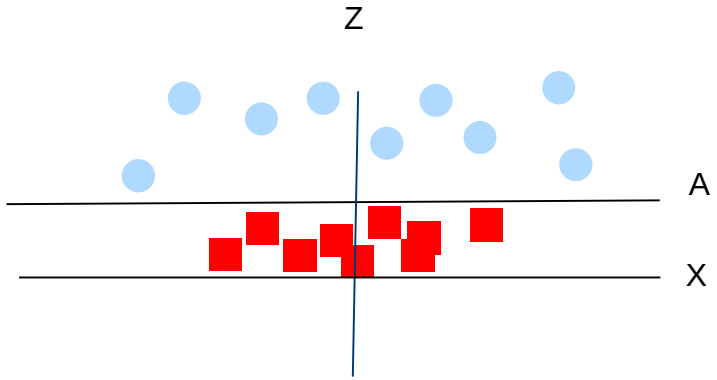


Support Vector Machine (continued)

- Here it would be impossible to draw a straight line that separates the groups, but we could translate this into a new feature z = x2 + y2



- The new feature 'z' could then be used to find the hyperplane using a built-in feature called kernel trick
  - Effectively it can solve for a non-linear feature



Apply SVM

- SVM looks like most other models and has lots of parameters to play with
  - Run it through several combinations to compare the results

```
from sklearn import svm
train_size = .03; test_size = .01
trainX, testX, trainY, testY = train_test_split( \
   df2.iloc[:,df2.columns != 'isFraud'], df2.isFraud, \
   train_size = train_size, test_size = test_size)

def do_SVM(kernel, gamma):
    print ("\nKernel:", kernel, "Gamma:", gamma)
    modelSVM = svm.SVC(gamma = gamma,  kernel = kernel)
    modelSVM.fit(trainX, trainY)
    print (modelSVM.score(testX, testY))
    predY = modelSVM.predict(testX)
    print (confusion_matrix(testY, predY))

do_SVM('linear', gamma='auto')
for kernel in ['rbf', 'poly', 'sigmoid']:
    for gamma in ['auto', 10, 100]:
        if not (kernel == 'poly' and gamma == 100):
           do_SVM(kernel, gamma)
```

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Classification Review

- Classification is one of the most widely used models
- It is supervised
- Good at predicting either/or or multiple-choice categories
- Lots of algorithms
- No one algorithm is best for all situations so often it involves running many of them, documenting the results, and choosing the best for your data and business case
- Can save the results of a lengthy training to a file and reload it for use with the predict function when needed

Next Steps

- Classification is one of the most useful models
- We have explored several different algorithms here and there are tons more, each with its own strengths and weaknesses
- Some other algorithms to explore:
  - Support Vector Machines
  - Boosted Trees
  - Random Forests
  - K-Nearest Neighbor
  - Stochastic Gradient Descent

Chapter Summary

In this chapter, we have:

- Understood the use cases for Classification models
- Discussed and compared various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network