BUSA3020-Week4-Lecture

February 7, 2023

BUSA3020 - Advanced Analytics Techniques

Week 4 Lecture - Classification Algorithms (Part 3)

Unit Convenor & Lecturer

George Milunovich
george.milunovich@mq.edu.au

References

1. Python Machine Learning 3rd Edition by Raschka & Mirjalili - Chapter 3 (second half)
2. Various open-source material

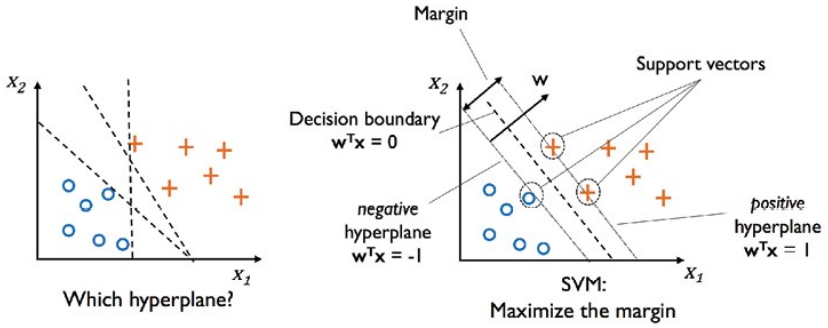
Week 4 Learning Objectives

- **Support Vector Machines (SVMs)**
  - Maximum Margin Classification
  - Dealing With a Nonlinearly Separable Case Using Slack Variables
  - Solving Nonlinear Problems Using Kernel SVM
  - Kernel Trick
  - Implementations in `scikit-learn`
- **Decision Tree Learning**
  - Maximizing Information Gain
  - Building a Decision Tree
- **Random Forests**
  - Combining Multiple Decision Trees

- **K-nearest Neighbors (KNN)**

Support Vector Machines (SVMs) - Maximum Margin Classification

Support Vector Machine can be seen as an extension of the perceptron.



A key concept in relation to SVMs is the **margin**. - The margin is related to the distance betwen the decision boundary (separating hyperplane) and the training examples that are closest to this boundary - Such closest examples are called **support vectors**

SVM attempts to **maximize the margin** thus obtaining maximal separation between the closest elements of two classes - The reason for preferring decision boundaries with large margins is that models with large margins are less likely to be overfitted - Large margins lead to better performance on test data (lower generalization error) - SVM is less sensitive to outliers than other classification algorithms because it mostly cares about the points closest to the decision boundary - In contrast the perceptron algorithm attempts to minimise misclassification errors

How SVM finds the maximum margin

First consider the case where we have only two features x_1 and x_2 and remember the equation of the straight line which we can use to plot the decision boundary in the above figure.

$w_1x_1 + w_2x_2 + w_0 = 0 \Rightarrow x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2}x_1$

Now consider the following two lines where the first line is on the positive (+) class side of the boundary and the second is on the negative (-) class side:

$w_1x_1^+ + w_2x_2^+ + w_0 = 1 \Rightarrow x_2^+ = \left(\frac{1}{w_2} - \frac{w_0}{w_2}\right) - \frac{w_1}{w_2}x_1^+$
$w_1x_1^- + w_2x_2^- + w_0 = -1 \Rightarrow x_2^- = \left(-\frac{1}{w_2} - \frac{w_0}{w_2}\right) - \frac{w_1}{w_2}x_1^-$

As you can see only the intercepts have changed. - The first equation is above the original line while the second equation is below the original line. - By choosing w_0, w_1 and w_2 we can change the intercept and slope of the decision boundary, as well as how far from it are the lines above and below (positive and negative hyperplanes).

How does SVM chose w_0, w_1 and w_2?

To find the distance between the two hyperplanes we subtract the last equation from the one above it:
$w_1(x_1^+ - x_1^-) + w_2(x_2^+ - x_2^-) = 2$

Next we divide the above equation by the length of the $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ which is defined as $||w|| = \sqrt{(w_1^2 + w_2^2)}$ to obtain the **distance** between the two hyperplanes which is the **margin**:

$\frac{w_1(x_1^+ - x_1^-) + w_2(x_2^+ - x_2^-)}{\sqrt{(w_1^2 + w_2^2)}} = \frac{2}{\sqrt{(w_1^2 + w_2^2)}} = \frac{2}{||w||}$

In order to maximize the margin SVM will do the following:

- Choose w_0, w_1, w_2 to minimize $\frac{1}{2}||w||^2$ subject to two **constraints**

1. $w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} \geq 1$ if $y^{(i)} = 1$ i.e. all positive-class examples fall above the positive hyperplane
2. $w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} \leq -1$ if $y^{(i)} = -1$ i.e. all negative-class examples fall below the negative hyperplane

This is typically done using quadratic programming and we will not get into mathematical details here.

Dealing with a nonlinearly separable case using slack variables

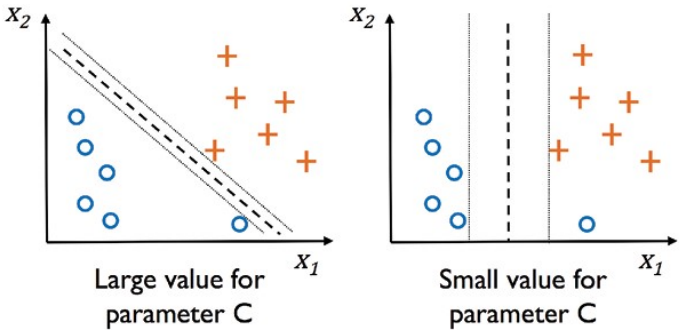
When dealing with nonlinearly separable data the above constraints will not be satisfied in the presence of misclassification - SVM will not be able to maximize the margin - Introduce **slack variables** $\xi > 0$ -> this is called **soft-margin clasification** - Slack variables will allow the algorithm to converge (find optimal w's) even when dealing with nonlinearly separable data and have classification errors

SVM will now solve do following problem:

- Minimize $\frac{1}{2}||w||^2 + C\left(\sum_i \xi^{(i)}\right)$ subject to the following constraints

1. $w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} \geq 1 - \xi^{(i)}$ if $y^{(i)} = 1$
2. $w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} \leq -1 + \xi^{(i)}$ if $y^{(i)} = -1$

Here C deals with how we treat misclassification and control the width of the margin - Large C -> Large error penalties for making misclassification errors - Decrease Bias - Increase Variance - Small C -> Small error penalties (less strict about making misclassification errors) - Increase Bias - Decrease Variance



`scikit-learn` SVM implementation

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(1)    # set random seed to be able to generate SAME random data every time

X_xor = np.random.randn(200, 2)

y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)

# print(np.hstack((X_xor[:10], y_xor[:10].reshape(-1,1))))
```

```
y_xor = np.where(y_xor, 1, -1)
print(np.hstack((X_xor[:10], y_xor[:10].reshape(-1,1))))


plt.scatter(X_xor[y_xor == 1, 0], X_xor[y_xor == 1, 1], c='b', marker='x', label='1') # columns
plt.scatter(X_xor[y_xor == -1, 0], X_xor[y_xor == -1, 1], c='r', marker='s', label='-1')

plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

[ ]:

[ ]:

[ ]:

[ ]:

To employ SVM we will use `scikit-learn` implementation - https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

- Lets also put our `plot_decision_regions` function from Week 3 into `plot_decision_regions.py` file

```
from sklearn.svm import SVC
import plot_decision_regions as pdr


svm = SVC(kernel='linear', C=1.0, random_state=1)
svm.fit(X_xor, y_xor)

pdr.plot_decision_regions(X_xor, y_xor, classifier=svm)

plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
print(f'Accuracy = {svm.score(X_xor, y_xor):.3f}')
```

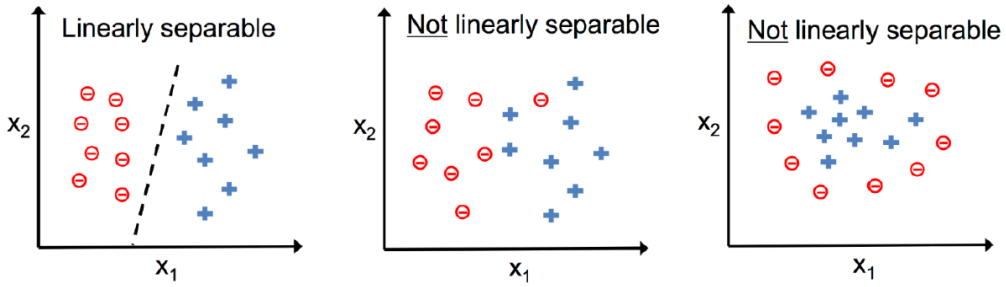
[ ]:

[ ]:

[ ]:

Linearly Inseparable Data and Kernel Methods

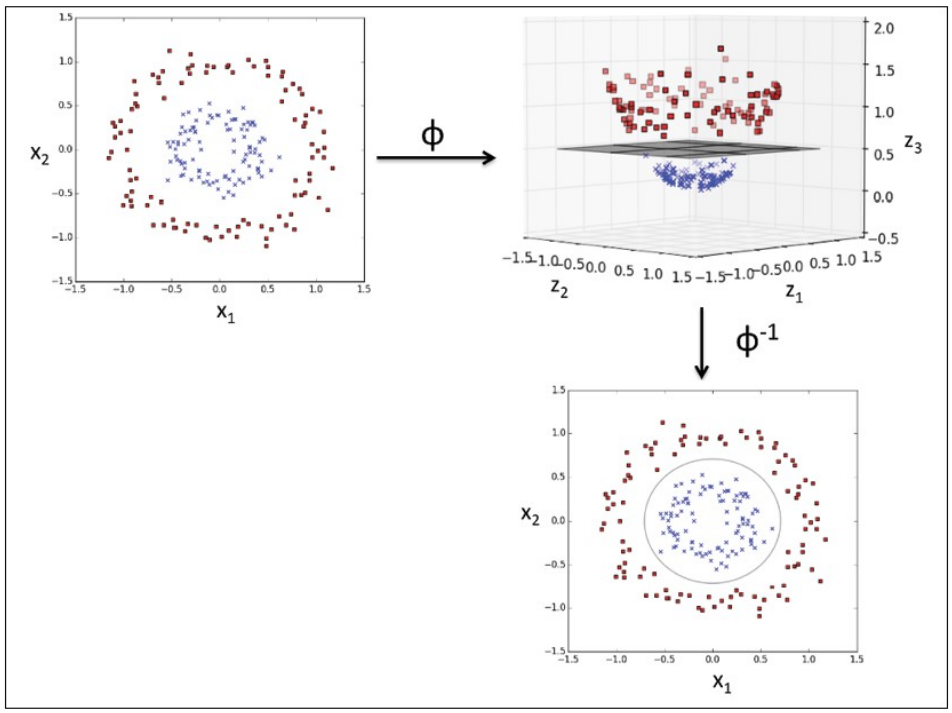
- As discussed before linearly inseparable data is data which is not perfectly separable using a line, a plane or a hyperplane



The above figure illustrates linearly inseparable data - None of the classification algorithms we have covered so far would be able to classify linearly inseperable data accurately

Kernel Methods - Deal with linearly inseparable data - Employ a nonlinear function ϕ - Create a nonlinear function of the orginal features using ϕ creating another dimension in the feature space - Data becomes linearly separable in the higher dimensional space - This allows us to separate the two classes via a linear hyperplane that becomes a nonlinear decision boundary when we project is back to the original feature space

Consider the following case



Here we have the following: - Orginal features are x_1 and x_2 - We can transform the original two-dimensional feature space into a three-dimensional feature space as follows - This can be done as follows $\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$ - the transformation we used here employs polynomial kernel, but there are many other functions which are used - a visualisation of this

example is here https://youtu.be/OdlNM96sHio

The Kernel Trick

The method we used above can be described as follows - Transform the orignial data using the function ϕ - This creates a higher dimensional feature space - Train a linear SVM to clasify the data in the new feature space - When we wish to classify test data transform it using ϕ as well - Apply trained SVM to tranformed test data

This approach is complicated and can be computationally expensive, especially if we have hundreds of features

- Most machine learning methods manipulate data using the dot (inner) product on feature vectors $x^{(i)^T}x^{(j)}$
  - remember that $x^{(i)^T}x^{(j)} = x_1^{(i)}x_1^{(j)} + x_2^{(i)}x_2^{(j)} + \cdots + x_m^{(i)}x_m^{(j)} = \sum_{\ell=1}^{m} x_\ell^{(i)}x_\ell^{(j)}$
- When tranforming data using ϕ instead of $x^{(i)^T}x^{(i)}$ we have $\phi(x^{(i)})^T\phi(x^{(j)})$
  - computing this part is computatinally expensive

Kernel Trick: instead of first choosing ϕ and computing $\phi(x^{(i)})^T\phi(x^{(j)})$ we define a **kernel function**

$\kappa\left(x^{(i)}, x^{(j)}\right) = \phi(x^{(i)})^T\phi(x^{(j)})$ directly

and use it instead of $\phi(x^{(i)})^T\phi(x^{(j)})$. The whole idea is that $\kappa\left(x^{(i)}, x^{(j)}\right)$ is easier to compute.

There are a number of kernel functions that we use in practice. For instance we have

Radial basis function (RBF) kernel also known as the **Gaussian kernel**

$\kappa\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\frac{||x^{(i)}-x^{(j)}||^2}{2\sigma^2}\right) = \exp\left(-\gamma||x^{(i)} - x^{(j)}||^2\right)$

where $\gamma = \frac{1}{2\sigma^2}$ is a hyperparameter - increasing γ will lead to a tighter and less smooth decision boundary

- Besides the RBF kernel there are many other kernel functions, some of which are implemented in scikit-learn
  - https://scikit-learn.org/stable/modules/metrics.html

Implementing a Nonlinear Kernel with SVM

- In our previous application of SVM above we used a `linear` kernel
  - Next we will use `rbf` kernel
- Consider two values for γ between 0.10 and 100
  - https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
- Later in the course we will learn how to find optimal values for hyperparameters such as γ, kernel and C

```
svm2 = SVC(kernel='rbf', gamma = 0.10, C=10.0, random_state=1)
svm2.fit(X_xor, y_xor)

pdr.plot_decision_regions(X_xor, y_xor, classifier=svm2)
```

```
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
print(f'Accuracy = {svm2.score(X_xor, y_xor):.3f}')

# ----------------------------------------

svm3 = SVC(kernel='rbf', gamma = 100, C=10.0, random_state=1)
svm3.fit(X_xor, y_xor)

pdr.plot_decision_regions(X_xor, y_xor, classifier=svm3)

plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
print(f'Accuracy = {svm3.score(X_xor, y_xor):.3f}')
```

[ ]:

[ ]:

Decision Tree Learning

- Decision trees can build complex decision boundaries by dividing the feature space into rectangles
- Decision tree model learns a series of questions regarding the features in the training set to infer the class labels of the examples
  - Here we have to be careful
  - The deeper a decision tree is, the more complex the decision boundary becomes, which can easily result in overfitting
- High interpretability of results
- Decision Trees (and Random Forests below) do not require feature scaling to be performed as they are not sensitive to the the variance in the data
- Consider deciding what to do on a particular day with the following class labels `{stay in, go to beach, go running, go to movies}`

How to build a decision tree

- Start at the tree root and split the data on the feature which results in the largest **information gain - IG**
- Repeat the splitting procedure at each child node until all training examples at each node belong to the same class (leaves are **pure**)
  - This can result in very deep trees with many nodes which are overfitted
- To avoid overfitting we set the maximal depth for the tree (we **prune** the tree)

Maximizing Information Gain (IG)

While each node can be split in many child nodes, most libraries (including scikit-learn) implements **binary decision trees** (each node has two child nodes).

We split the nodes at most informative features by optimizing an objective function

$IG(D_p, f) = I(D_p) - \frac{N_{\text{left}}}{N_{\text{p}}} I(D_{\text{left}}) - \frac{N_{\text{right}}}{N_{\text{p}}} I(D_{\text{right}})$

- f - feature to perform the split, e.g. age or education level
- D_p - dataset of the parent node
- N_p - number of training examples at the parent node
- D_j - dataset of the jth child node
- N_j - number of training examples in the jth child node
- I - Impurity measure

Information Gain (IG) is the difference between the impurity of the parent node and the sum of the child node impurities - The node impurity is a measure of the homogeneity (sameness) of the labels at the node - The lower the impurities of the child nodes the larger the information gain

Let $p(i|t)$ be the proportion of the examples that belong to class i for a node t - E.g. $p(i = 1|t) = 0.5$ and $p(i = 0|t) = 0.5$ - Let there be c number of classes in total, e.g. $c = 2$ as above

We typically use 3 types of impurity measure: 1. **Entropy** - $I_H = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$ - Maximal if classes are perfectly mixed - E.g. if $p(i = 1|t) = 1$ in which case $p(i = 0|t) = 0 \Rightarrow I_H = -(1(0) + 0(-\infty)) = 0$
- E.g. if $p(i = 1|t) = 0.5$ in which case $p(i = 0|t) = 0.5 \Rightarrow I_H = -\left(0.5\log_2(0.5) + 0.5\log_2(0.5)\right) = 1$

2. **Gini Impurity**

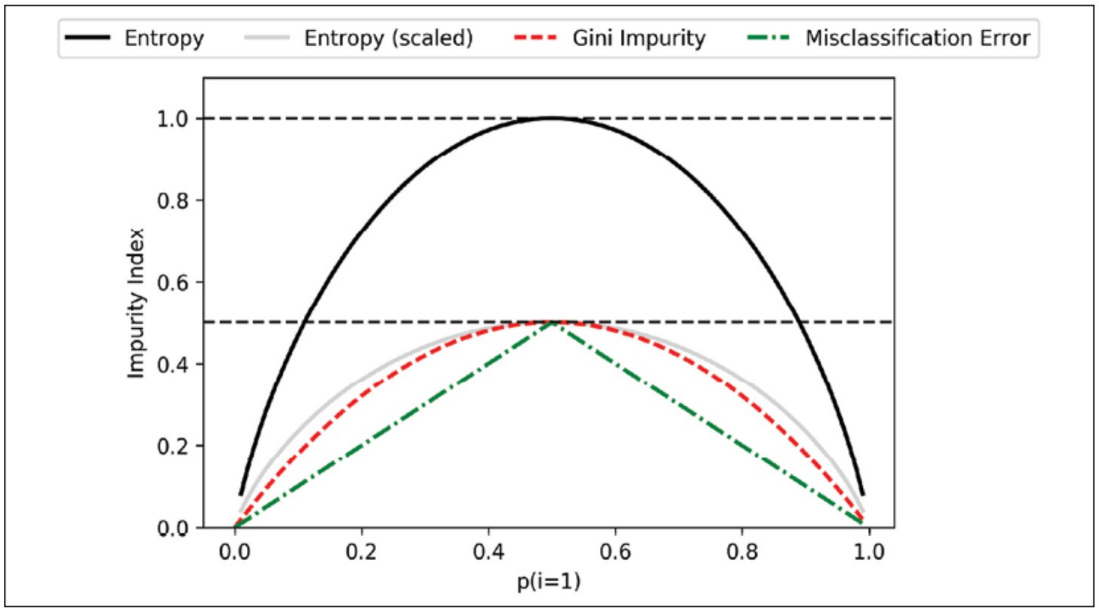
- $I_G = \sum_{i=1}^{c} p(i|t)(1 - p(i|t))$
- Maximal if classes are perfectly mixed
- E.g. if $p(i = 1|t) = 1$ in which case $p(i = 0|t) = 0 \Rightarrow I_G = (1(0) + 0(1)) = 0$

- E.g. if $p(i = 1|t) = 0.5$ in which case $p(i = 0|t) = 0.5 \Rightarrow I_G = (0.5(0.5) + 0.5(0.5)) = 0.5$

3. **Classification Error** (I_E)

- $I_E = 1 - \max[p(i|t)]$
- Maximal if classes are perfectly mixed
- Less sensitive to changes in class probabilities of the nodes
- E.g. $p(i = 1|t) = 1$ in which case $p(i = 0|t) = 0 \Rightarrow I_E = 1 - 1 = 0$

- E.g. $p(i = 1|t) = 0.5$ in which case $p(i = 0|t) = 0.5 \Rightarrow I_E = 1 - 0.5 = 0.5$



Lets consider splitting a parent node which has (40, 40) examples, i.e 40 examples from class 1 and 40 examples from class 2, in two different ways - A: left node: (30, 10), right node: (10, 30) - B: left node: (20, 40), right node: (20, 0)

Now we compare the two splits A & B based on the three impurity measure - Note that B split is purer

- Classification Error
  - IG = 0.25 under both scenarios
  - Make sure you can do these computations
- Gini Impurity
  - Gini impurity favours B split (IG = 0.16) over A split (IG = 0.125)
  - Make sure you can do these computations
- Entropy
  - Entropy favours B split (IG = 0.31) over A split (IG = 0.19)
  - Make sure you can do these computations

$$I_H(D_p) = -\left(0.5\ log_2(0.5) + 0.5\ log_2(0.5)\right) = 1$$

$$A: I_H(D_{left}) = -\left(\frac{3}{4}\ log_2\left(\frac{3}{4}\right) + \frac{1}{4}\ log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A: I_H(D_{right}) = -\left(\frac{1}{4}\ log_2\left(\frac{1}{4}\right) + \frac{3}{4}\ log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19$$

$$B: I_H(D_{left}) = -\left(\frac{2}{6}\ log_2\left(\frac{2}{6}\right) + \frac{4}{6}\ log_2\left(\frac{4}{6}\right)\right) = 0.92$$

$$B: I_H(D_{right}) = 0$$

$$B: IG_H = 1 - \frac{6}{8}0.92 - \ 0 = 0.31$$

Building a decision tree with `scikit-learn`

- `scikit-learn` provides https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
- As we can see there are a number of parameters to set such as `criterion` and `max_depth`

Lets apply a decision tree to our linearly inseparable data generated above - set `max_depth = 4` & `criterion='gini'`

```
from sklearn.tree import DecisionTreeClassifier

tree_model = DecisionTreeClassifier(criterion='gini', max_depth = 4, random_state=1)
tree_model.fit(X_xor, y_xor)
```

```
pdr.plot_decision_regions(X_xor, y_xor, classifier=tree_model)

plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

```
print(f'Accuracy = {tree_model.score(X_xor, y_xor):.3f}')
```

[ ]:

[ ]:

Lets visualize the decision tree model

```
from sklearn import tree

plt.style.use('seaborn')
tree.plot_tree(tree_model)
plt.savefig('tree1.png')

plt.show()
```

[ ]:

Alternatively we can use `Graphviz` program to produce a nicer image

We will need to install three new packages using the following commands

```
conda install pydotplus
conda install graphviz
conda install pyparsing
```

Restart Anaconda Navigator (do not just close browser window).
We should now be able to use `graphviz` to plot our fitted tree model

```
from pydotplus import graph_from_dot_data
from sklearn.tree import export_graphviz
from IPython.display import Image

dot_data = export_graphviz(tree_model, filled=True, rounded=True, class_names=['-1', '+1'], fea
graph = graph_from_dot_data(dot_data)
graph.write_png('tree.png')


Image(graph.create_png())
```

[ ]:

[ ]:

[ ]:

Random Forests

Random Forests are **ensembles** of decision trees - Average multiple decision tree which may individually suffer from high variance (overfitting) - The process of averaging may reduce the

amount of overfitting and result in a model which generalizes better to unseen data - More difficult to interpret

Random Forest Algorithm 1. Draw a random **bootstrap** sample of size n from the training dataset (with replacement) 2. Grow a decision tree from the bootstrap sample: - Randomly select d features without replacement (different from just training one tree) - Build a tree using these d features 3. Repeat 1.-2. k times 4. Aggregate the prediction by each tree to assign the class label by **majority vote** - Every tree makes a prediction (votes) for each test example and the final output prediction is the one that receives more than half of the votes. If none of the predictions get more than half of the votes, we may say that the ensemble method could not make a stable prediction for this instance.

Hyperparameters - number of trees (k) - typically the larger the k the better the perfmance but more computationally expensive - bootstrap size (n) - smaller n -> increase randomness of random forest -> can reduce overfitting but also increase bias - `scikit-learn` sets n =sample size of training set (but with replacement) - number of random features used, `scikit-learn` sets $d = \sqrt{m}$, where m is the total number of features

`scikit-learn` implementation: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

---

Fitting Random Forest with `scikit-learn`

```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(criterion='gini', n_estimators=25, random_state=1, n_jobs=2)
forest.fit(X_xor, y_xor)


pdr.plot_decision_regions(X_xor, y_xor, classifier=forest)
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
print(f'Accuracy = {forest.score(X_xor, y_xor):.3f}')
```

[ ]:

[ ]:

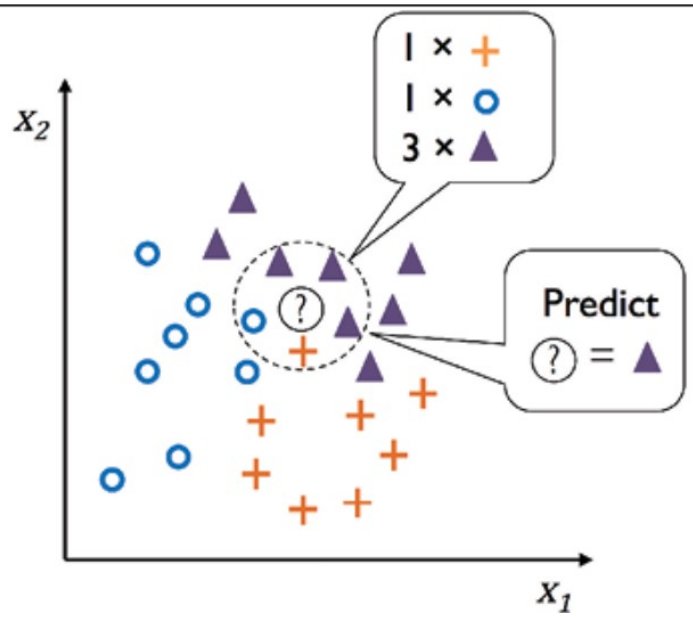
K-nearest Neighbors (KNN)

KNN algorithm finds k examples in the training set that are closest to the point we try to classify according to a distance metric chosen.

KNN is a **lazy learner** -> it does not learn a decision boundary using some function of data but instead memorizes the training dataset

KNN Algorithm 1. Choose a number k and a distance metric 2. Find the k-nearest neighbors of the data example we need to classify 3. Assign the class label by majority vote

Example: classifying a new datapoint (?) based on 5 nearest neighbors from the training set



Hyperparameters - k - crucial in finding a good balance between overfitting and underfitting - Distance metric -> different metrics will find different neighbors

---

Lets find the distance betwen two vectors $x^T = [x_1 \quad x_2 \quad ... \quad x_n]$ and $z^T = [z_1 \quad z_2 \quad ... \quad z_n]$ - Minkowski distance (metric) is equal to $d(x,z) = \sqrt[p]{\sum_{i=1}^{n} |x_i - z_i|^p}$

- When $p = 2$ we have $d(x,z) = ||x - z|| = \sqrt{\sum_{i=1}^{n}(x_i - z_i)^2}$ - Setting $z = 0$ (vector zero) will give us the norm (vector length) of x from before

KNN advantage: immediately adapts as we collect new data
KNN disadvantage: computationally complexity grows as we add more examples to our training set

`scikit-learn` https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

- Try varying `n_neighbors` over across the values {50, 20, 2}

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')

knn.fit(X_xor, y_xor)


pdr.plot_decision_regions(X_xor, y_xor, classifier=knn)

plt.legend(loc='upper left')
```

```
plt.tight_layout()
plt.show()
print(f'Accuracy = {knn.score(X_xor, y_xor):.3f}')
```

[ ]:

[ ]:

[ ]: