# Machine Learning with Pytorch and Scikit-Learn

**Questions**

**Helpful Links:**

* [GitHub Repository](https://github.com/rasbt/machine-learning-book)

**Ideas**

**Notes**

**Matrix Notation**

* –> superscript represents the row, subscript represents the column
* A data set consisting of 150 examples of 4 features, can be written as a 150x4 matrix, formally denoted as
* Superscript “i” to refer to the “ith” training example
* Subscript “j” to refer to the “jth” dimension of the training dataset
* Vector = lowercase bold letter
* Matrix = uppercase bold letter

# Ch 1: Giving Computers the Ability to Learn from Data

## The 3 Types of Machine Learning

* Supervised learning
  + Labeled data
  + Direct feedback
  + Predict outcome/failure
* Unsupervised learning
  + No labels/targets
  + No feedback
  + Find hidden structure in data
* Reinforcement learning
  + Decision process
  + Reward system
  + Learn series of actions

## Supervised Learning

* Two examples of subcategories of supervised learning: (not exhaustive)
  + Classification
  + Regression
* Classification is for discrete categorical variables
* Regression is for continuous variables

## Reinforcement Learning

* The algorithm will learn to maximize a set reward function
  + Learning a series of decisions that will maximize the reward function
  + Ex: learning to play chess and the reward function is for “win %”

## Unsupervised Learning

* Dealing with unlabeled or unstructured data, able to extract meaningful information without the guidance of known outcome variable or reward function
* Subfields of unsupervised learning (not exhaustive):
  + Clustering
  + Dimensionality reduction –> compresses data onto a smaller dimensional subspace while retaining the most important information

## The roadmap for building ML systems

Diagram

Description automatically generated

* **Preprocessing – getting data into shape:**
  + Many ML algorithms require that the selected features are on the same scale for optimal performance –> often achieved by transforming the features in the range [0,1] or a standard normal distribution with zero mean and unit variance
    - Formula to put data into a standard normal distribution:

Graphical user interface

Description automatically generated with low confidence

* + - * The “~” means that random variable “X” is distributed according to the Gaussian distribution (noted with the “N”) of (mean, variance)
      * <https://stats.stackexchange.com/questions/248776/can-someone-explain-intuitively-what-does-0-mean-and-unit-variance-mean>
  + Some of the selected features may be highly correlated (ie. Somewhat redundant), dimensionality reduction can be useful for compressing the features onto a lower-dimensional subspace
    - Advantages of reducing dimensional subspace (not exhaustive):
      * Less storage is required –> algorithm can run faster
      * Can sometimes, improve predictive performance if the dataset contains a large amount of irrelevant features (ie. Low signal-to-noise ratio)
  + Want to split dataset and into training set and test set –> will use test set to see how well model performs after trained
* **Training and selecting a predictive model:**
  + Each ML algorithm has its own inherent assumptions/biases –> should compare a handful of algorithms and select the one with the best performance
  + To compare models, need a metric to judge comparison –> classification accuracy is commonly used
    - Classification accuracy - proportion of correctly classified instances
  + How do we know which model performs well on the final test dataset and real-world data if we don’t use this test dataset for the model selection, but keep it for model evaluation?
    - Cross-validation can be used to address this issue embedded in the question
      * Divides dataset into training and validation subsets in order to estimate the generalization performance of the model
  + Can’t expect the default hyperparameters of libraries to be optimal for your given problem task –> need to adjust accordingly
* **Evaluating models and predicting unseen data instances**
  + After selecting model that has been fitted on the training dataset, can use the test dataset to estimate model performance on unseen data (to estimate “generalization error”)
    - Are applying the same parameters from the training set to the test set

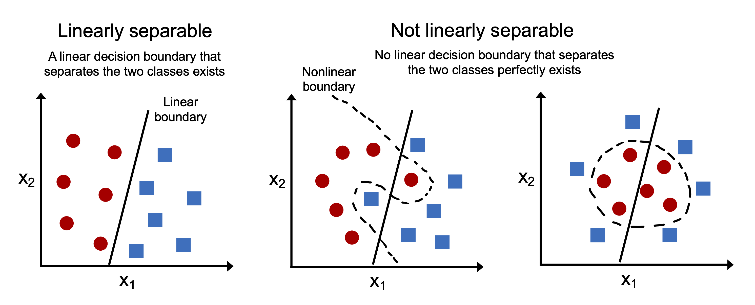
# Ch 2: Training Simple Machine Learning Algorithms for Classification

## The Formal Definition of an Artificial Neuron

* A decision function, , that takes a linear combination of certain input values, “x”, and a corresponding weight vector, “w”, where “z” is the net input –>
  + ,
* If the net input, “z”, of a particular example, , is greater than a defined threshold function, , we predict class 1 and class 0 otherwise:
* Can make do some transformations to alter the equation to:
  + –>
    - “b” is defined as negative theta –> that way if “z” + “b” >= 0 then 1, otherwise 0

## The Perception Learning Rule

* Either perceptron fires or it doesn’t –> binary
* Perceptron algorithm summary:
  + 1) initialize the weights and bias unit to 0 or small random #s
  + 2) for each training example, :
    - A) compute the output value,
    - B) update the weights and bias unit
* Each iteration reupdate weights and bias:
  + ; and
* The update value deltas are computed as:
  + ; and
    - ^^^ note that each weight corresponds with a feature (“j” in the above), but the bias does not
    - Eta = the learning rate (typically a constant between 0.0 and 1.0)
    - = true class label
    - = predicted class label
  + All the weights and biases are being updated simultaneously –> I think that means that each predication is made for 1st to the ith and then the weights are updated
* With the above equations the larger the misclassification, the larger the change to the weight and biases will be to counteract that next time around:
  + Assume:
  + Assume: and we misclassify this example as class 0 ^^^ –> would increase corresponding weight by 2.5 in total so that the net input would be more positive the next time, and thus more likely to be above the threshold of the unit step function to classify the example as class 1
    - = 1.5
    - = 1
* The convergence of the perceptron is only guaranteed if the 2 classes are linearly separable –> if not, the perceptron would never stop updating no matter how many epochs because it could never converge on the linear separation –> there are ways to deal with this



## Implementing a Perceptron Learning Algorithm in Python

* **Choosing the correct weights upon initialization:**
  + If the initial weight vector is zero, the algorithm will not be able to learn
    - <https://www.deeplearning.ai/ai-notes/initialization/>
* Side note on OvA method for multi-class classification:
  + OvA (one-versus-rest (OvR)), is a technique that allows us to extend any binary classifier to multi-class problems
  + Can train one classifier per class, where the particular class is treated as the positive class and the examples from all other classes are considered negative classes

## Adaptive linear neurons and the convergence of learning

* Another type of single-layer NN: ADAptive LInear NEuron –> Adaline
* Key difference between Adaline (aka Widrow-Hoff rule) and Rosenblatt’s perceptron:
  + The weights are updated based on a linear activation function rather than a unit step function like in the perceptron
    - The linear activation function is used for learning the weights, we still use a threshold function to make the final prediction –> similar to the unit step in Rosenblatt’s perceptron

Diagram, schematic

Description automatically generated

## Minimizing loss functions with gradient descent

* We want to minimize the objective function (loss function, cost function, etc)
* For Adaline the loss, L, is defined:
  + - ­­The term ½ is just added for our convenience and will make it easier to derive the gradient of the loss function with respect to the weight parameters
  + The main advantage of this continuous linear activation function, in contrast to the step function, is that the loss function is differentiable
    - It is also convex –> which means we can use gradient descent to minimize the loss function
* Gradient descent:
  + In each iteration, we take a step in the opposite direction of the gradient
    - The step size is determined by the learning rate and the slope of the gradient
* With that said, some equation alterations:
  + We will take a step in the opposite direction of the gradient, , of our loss function,
  + ;
  + The parameter changes, , are defined as the negative gradient multiplied by the learning rate, eta:
    - ;
  + To compute the gradient of our loss function, we need to compute the partial derivative of our loss function with respect to each weight, :
  + Similarly compute the partial derivative of the loss function with respect to the bias:
  + Remember the “2” in the numerator above is merely a constant scaling factor and we could omit it without affecting the algorithm –> it would have the same impact as changing the learning rate by a factor of 2
  + Weight update:
    - ;
  + Since we update all parameters simultaneously, our Adaline learning rule becomes:
    - ;
    - ^^^ This looks identical to the perceptron rule, however, note that:
      * with , is a real number and not an integer class label
      * Furthermore, the weight update is calculated based on all class examples in the training dataset (instead of updated parameters incrementally after each training example) –> will refer to this as ***full batch gradient descent***
* **Side note, explanation of what nabla,** **, means:**
  + <https://math.stackexchange.com/questions/2710328/what-does-the-symbol-nabla-indicate>

**Implementing Adaline in Python**

* Learning rate and the # of epochs are hyperparameters

## Improving gradient descent through feature scaling

* Use feature scaling to try and get optimal performance out of our algorithms –> feature scaling is broad
* One method of feature scaling is ***standardization***
* Below is a way to standardize “j” feature:
  + This normalization procedure helps gradient descent converge more quickly –> easier to find learning rate that works well for all weights (and the bias)
  + It does not make the original dataset normally distribute –> it shifts the mean of each feature so that it is centered at 0, with a standard deviation of 1 (unit variance)

## Large-scale ML and stochastic gradient descent

* If our dataset is huge, doing full-batch gradient descent can be computationally expensive since you need to reevaluate the whole training dataset each time you take one step toward the global minimum
* ***Stochastic Gradient Descent (SGD)*** is a popular alternative to the batch gradient descent algorithm
* Instead of updating the weights based on the sum of the accumulated errors over all training examples, like batch does, we update the parameters incrementally for each training example:
  + ;
* Although SGD can be considered an approximation of gradient descent, it typically reaches convergence quicker because of the frequent weight updates
* The frequent weight updates also cause more noise, but this can help to escape local minima if working with nonlinear loss functions
* When using SGD, NEED to present the data in a random order so that the algorithm doesn’t learn cycles –> will shuffle the training data after every epoch
* Another advantage of SGD is that it can be used for ***online learning*** –> it can easily adapt to new data and you can discard the old data after its been used if storage is the issue
* **Adjusting the learning rate during training:**
  + SGD typically replaces the fixed learning rate with an adaptive learning rate that decreases over time:
    - –> where c\_1 and c\_2 are constants
  + SGD doesn’t reach the global minimum but an area very close to it
* Mini-batch gradient descent is kind of a halfway point between batch gradient descent and SGD

# Ch 3: A Tour of Machine Learning Classifiers using Scikit-Learn

## Choosing a Classification Algorithm

* Each algorithm has its own quirks and is based on its own assumptions
  + No free lunch theorem 🡪 no single algorithm works best across all scenarios
* 5 main steps involved in training a supervised ML algorithm:
  + 1) Selecting features and collecting labeled training examples
  + 2) choosing a performance metric
  + 3) choosing a learning algorithm and training model
  + 4) evaluating the performance of the model
  + 5) changing the settings of the algorithm and tuning the model

## Modeling Class Probabilities via Logistic Regression

* The biggest disadvantage of the perceptron algorithm is that it never converges if the classes are not perfectly linearly separable
  + The classification example in the chapter is an example
    - The reason is that the weights are being consistently updated because there is always at least one misclassified example
* Logistic regression is great for linear and binary classification
  + Despite its name 🡪 is a model for classification NOT regression

## Logistic Regression and Conditional Probabilities

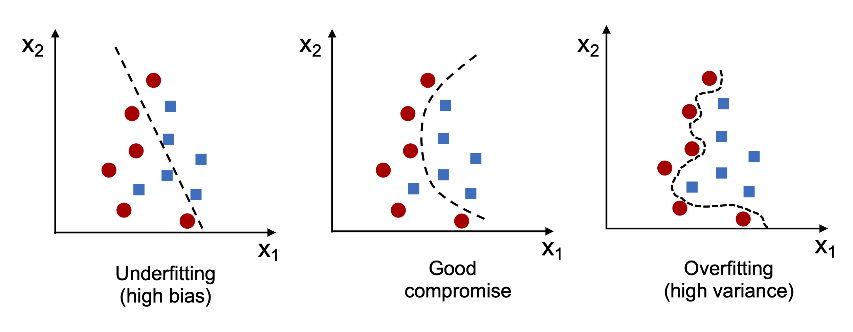
* ***Sidenote:*** Logistic Regression for Multiple Classes:
  + Logistic regression can be generalized to multiclass settings 🡪 multinomial logistic regression or SoftMax regression or can use the OvR technique
* Odds of an event 🡪 … where ‘p’ stand for probability of a particular event
* Conditional probability: 🡪 probability that an example belongs to the ‘y’ class 1, given its feature ‘x’
* Can define the **logit function,** simply the logarithm of the odds (log-odds):
  + - Log refers to the natural logarithm
  + Logit function takes input values in the range 0 to 1 and transforms them into values over the entire real-number range
    - The inverse of the logit function will map the real-number range back to a [0, 1] range for the probability ‘p’
    - Inverse of the logit function is typically called the ***logistic sigmoid function*** (sigmoid function):
      * 🡪 net inputs 🡪 linear combination of weights, and the inputs
      * As ‘z’ increases sigma(z) will approach 1 🡪 e^-z will become increasingly small
        + And vice versa
* We assume that there is a linear relationship between the weighted inputs (net inputs in chapter 2) and the log-odds:
* Sigmoid function is your activation function and then will use a threshold function to determine the class membership of the example
  + Can also just use the sigmoid function to determine class probability

## Learning the Model Weights via the Logistic Loss Function

* In prior chapter, defined the mean squared error loss function as follows:
* To explain how to derive loss function for logistic regression, let’s first define the likelihood, “L”, that we want to maximize when we build a logistic regression model
  + Assuming the individual dataset examples are independent of one another, the formula is as follows:
  + In practice can be easier to maximize using the (natural) log of this equation (log-likelihood):
* We could use gradient-ascent to maximize the log-likelihood. Alternatively, will rewrite function so that we can minimize using gradient-descent:
* To get a better grasp of the loss function, let’s look at it for a single training example:
  + So, the first term becomes 0 if y=0 and the 2nd term becomes zero if y=1:

## Tackling Overfitting via Regularization

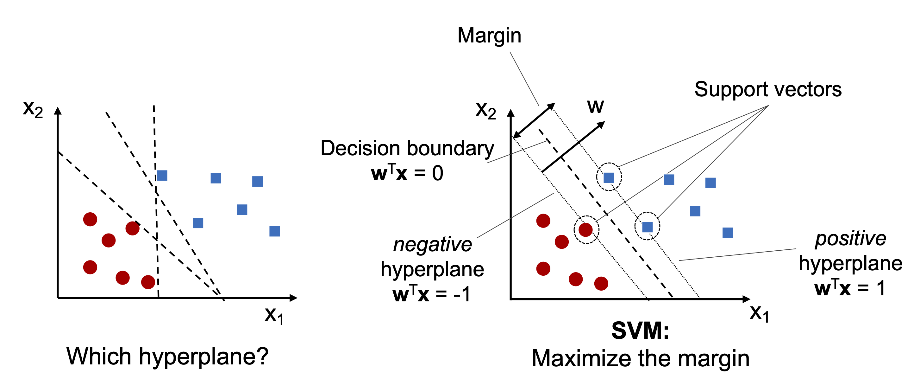
* Regularization is the process of calibrating your model to prevent over/underfitting
* Two common methods of regularization are the ridge and lasso method:
  + both multiply a variable "lambda" by the error formula --> the "lambda" is an adjustable multiple that you set for the penalty for errors
  + can increase/decrease to adjust how you want to penalize the errors
  + the ***ridge method*** uses the squared sum of the coefficient errors
  + the ***lasso method*** uses the absolute value sum of the coefficient errors
    - With that being said if you consider the difference between 5 and 10 when both are squared vs absolute value... the ridge method will magnify (have a higher penalty) for larger errors when compared to the lasso method
  + <https://www.simplilearn.com/tutorials/machine-learning-tutorial/regularization-in-machine-learning#what_is_regularization_in_machine_learning>
* Overfitting – Will perform well on training data but won’t be able to generalize well (called high variance)
  + Underfitting is called high bias
  + **Hence, the bias-variance trade-off**



* Regularization is a useful method for handling collinearity (high correlation among feature) filtering out noise from data, and eventually preventing overfitting
  + The concept behind regularization is to introduce additional information to penalize extreme parameter (weights) values
* Common form of regularization is call ***L2 Regularization*** (aka: L2 shrinkage, weight decay):
  + ^^^ “lambda” is the so-called regularization parameter
  + 2 as the denominator is merely a scaling factor, such that it cancels when computing the loss gradient
* Regularization is another reason why feature normalization is important. For regularization to work properly, we need to ensure that all the features are on comparable scales
* Can simply add the regularization term to the end of the loss function to shrink the weights during model training:
* This would change the partial derivative of the loss function to the following:
* We can then throttle the regularization parameter “lambda” to control how closely we fit the training data
  + **Note:** the bias unit, which is essentially an intercept term or negative threshold, is usually not regularized
* Increasing the regularization strength can reduce overfitting. But, if it is too high and the weight coefficients approach zero, the model can perform poorly due to underfitting

## Maximum Margin Classification w/ Support Vector Machines (SVMs)

* Support vector machines (SVM) are another powerful and widely used algorithm, which can be considered an extension of the perceptron
  + Using the perceptron, we minimized misclassification errors
  + However, in SVMs, our optimization objective is the maximize the margin
    - Margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called support vectors



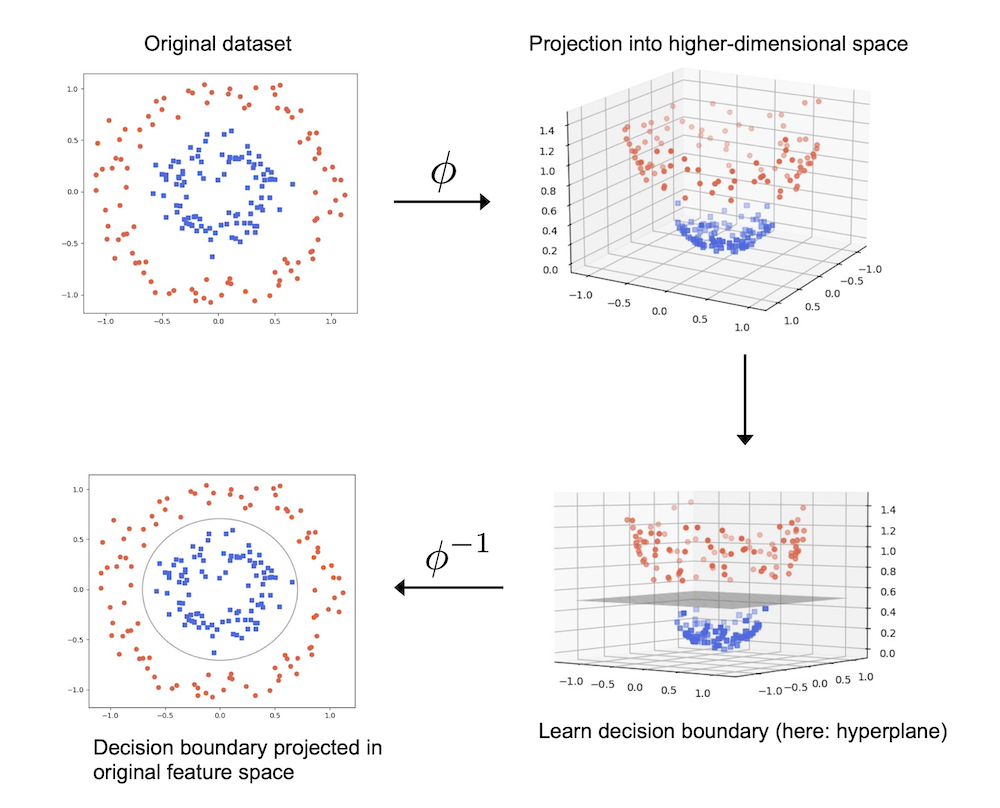
* + - ^^^ essentially saying that you want it to be as clear as possible which class each instance belongs to
* Rational behind having decision boundaries with large margins is that they tend to have a lower generalization error (less prone to overfitting)
* Mathematics behind SVMs relates to constrained optimization 🡪 can look if so desired

## Dealing with a Nonlinearly Separable Case Using Slack Variables

* Slack variable and soft-margin classification
* Motivation for the slack variable was that the linear constraints in the SVM optimization objective need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate loss penalization
* The use of the slack variable introduces a “C” variable which we can increase or decrease to control the penalty for misclassifications
  + “C” variable controls the width of the margin to the decision boundary and therefore the bias-variance tradeoff

## Solving Nonlinear Problems Using a Kernel SVM

* Another reason why SVMs enjoy high popularity among ML practitioners is that they can be easily kernelized to solve nonlinear classification problems
* The basic idea behind kernel methods for dealing with linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher-dimensional pace via a mapping function, ɸ, where the data becomes linearly separable
  + Example of transforming a 2-D dataset into a 3-D feature space:



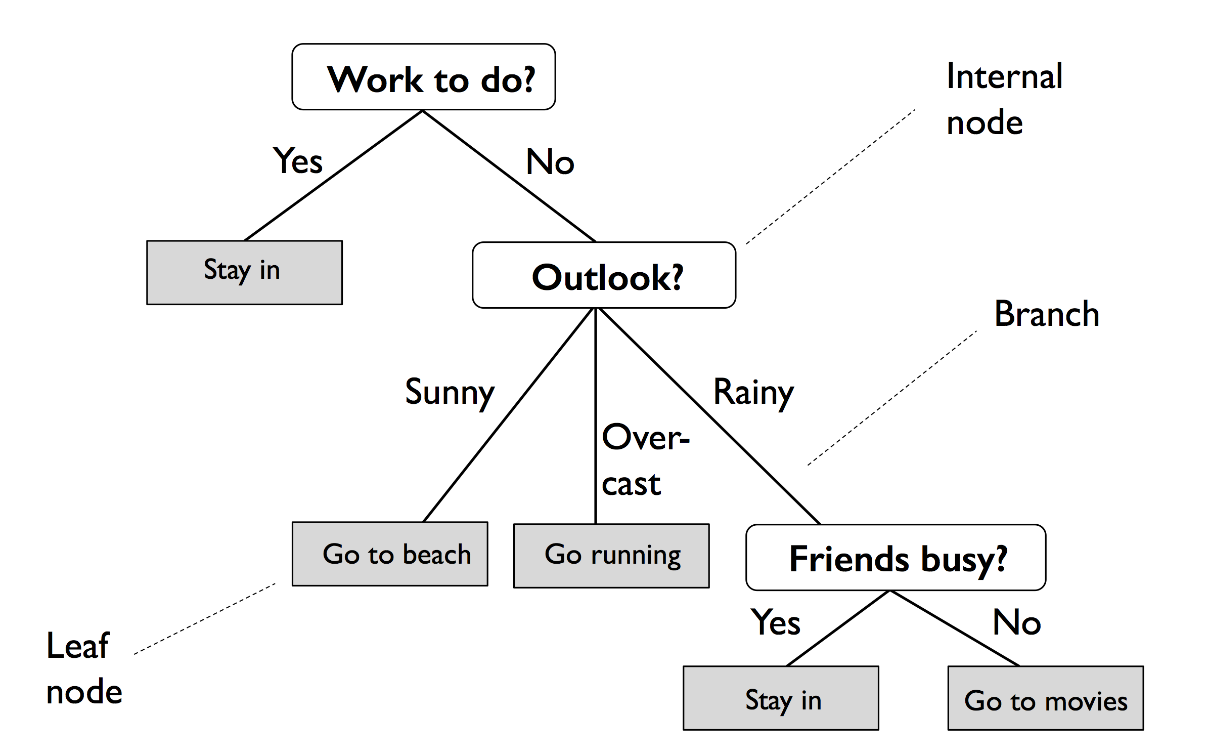
* + The equation allows us to separate the classes via a linear hyper plane that becomes a nonlinear decision boundary

## Using the Kernel Trick to Find Separating Hyperplanes in a High-Dimensional Space

* To solve a nonlinear problem using an SVM:
  + 1) transform training data into a higher-dimensional space via a mapping function (“phi”)
  + 2) train a linear SVM model to classify the data in this new feature space
  + 3) use same mapping function (“phi”) to transform the unseen test data and then use model to classify
* ^^^ this can be computationally expensive and is where the ***kernel trick*** becomes useful
* Haven’t gone into much detail about how to solve the quadratic programming task to train an SVM, in practice, just need to replace the dot product by
  + To save the expensive step of calculating this dot product between two points explicitly, we define a so-called ***kernel function***:
  + One of the most widely used kernels is the ***radial basis function (RBF)*** kernel, which can simply be called the Gaussian kernel: 🡪 simplified to
    - In the above simplification, is a free parameter to be optimized
* The term “kernel” can be interpreted as a ***similarity function*** between a pair of examples
  + The minus sign inverts the distance measure into a similarity score, and, due to the exponential term, the resulting score will fall into a range between 1 (for exactly similar examples) and 0 (for very dissimilar examples)
* Good summary on SVMs, kernels, and RBF: <https://towardsdatascience.com/svm-classifier-and-rbf-kernel-how-to-make-better-models-in-python-73bb4914af5b>

## Decision Tree Learning

* Decision tree classifiers are attractive models if we care about interpretability



* The above shows decision tree for categorical variables but the same can apply for quantitative variables (ex: “petal width > 2.8cm”)
* Using the decision tree algorithm, we start at the tree root and split the data on the feature that results in the largest ***information gain (IG)***
  + In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure (the training examples at each node all belong to the same class, aka no misclassifications)
    - This can result in a very deep tree with many nodes, which can lead to overfitting. Thus, we typically want to ***prune*** the tree by setting a limit for the maximum depth of the tree

## Maximum IG – getting the most bang for your buck

* Objective function to maximize the IG at each split:
  + - “f” is the feature to perform the split
    - “D\_p” and “D\_j” are the dataset of the parent and the jth child node
    - “I” is our impurity measure
    - “N\_j” is the # of examples of the jth child node
  + IG is simply the difference between the impurty of the parent node and the sum of the child node impurities--the lower the impurities of the child node, the large the IG
* However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees 🡪 parent node is split into 2 child nodes (D\_left and D\_right):
* 3 common impurity measures or splitting criteria for decision trees:
  + ***Gini impurity*** (I\_G)
  + ***Entropy*** (I\_H)
  + ***Classification error*** (I\_E)

**Impurity Measure: Entropy (IH)**

* Tries to maximize the mutual information in the tree
* For all non-empty classes (p(i|t) != 0):
  + p(i|t) = proportion of examples that belong to class “i” for a particular node “t”
    - entropy is 0 if all examples belong to the same class

**Impurity Measure: Gini (IG)**

* Tries to minimize the probability of misclassification
* Similar to entropy, the Gini impurity is maximal if the classes are perfectly mixed 🡪 in a binary setting (c = 2):
  + 🡪 would be 0.25 but c = 2 so do it twice and sum
* In practice Gini and entropy yield very similar results and the ***pruning cutoffs*** tend to be the more important hyperparameter to experiment with

**Impurity Measure: Classification Error (IE)**

* Useful for pruning, but not recommended for growing a decision tree 🡪 since it is less sensitive to changes in class probabilities of nodes

## Combining multiple decision trees via random forests

* An ensemble method 🡪 combining methods (will cover more in chapter 7)
* Random forest – known for good scalability and ease of use 🡪 can be considered an ensemble of decision trees
* Random forest algorithm summarized:
  + 1) draw a random bootstrap sample of size “n” (randomly choose “n” examples from the training dataset w/o replacement)
  + 2) grow a decision tree from the bootstrap sample. At each node:
    - Randomly select “d” features w/o replacement
    - Split the node using the feature that provides the best split according to the objective function (ex: maximizing IG)
  + 3) repeat steps 1-2 “k” times
  + 4) aggregate the prediction by tree to assign the class label by majority vote
* Random forests don’t offer as good of interpretability as decision trees but you typically don’t have to worry as much about good hyperparameter values 🡪 the number of trees, “k”, is the most important hyperparameter
  + Other important hyperparameter are the size, “n”, and the # of features, “d”
    - Sample size, “n”, controls the bias-variance tradeoff 🡪 lower sample size of each true would increase randomness and decrease overfitting
  + For choosing the # of features “d”, a reasonable default is d = sqrt(m) 🡪 where “m” is the # of total features in the training set

## K-nearest neighbors – a lazy learning algorithm

* The algorithm is:
  + Supervised
  + “lazy learner” because it doesn’t learn a discriminative function from the training data but memorizes the dataset instead
  + Non-parametric model 🡪 not characterized by fixed parameters, and the # of parameters changes with the amount of training data
  + Susceptible to the curse of dimensionality
* Summary of KNN steps:
  + 1) choose the # of “k” and a distance metric
  + 2) find the k-nearest neighbors of the data record we want to classify
  + 3) assign the class label by majority vote
* The upside of the KNN, a memory based approach, is that it incorporates the new data of each additional training datapoint. However, the computational intensity grows linearly.
* The right choice of “k” is crucial for the bias-variance tradeoff
* Distance metric needs to be appropriate for the features in the dataset
  + Euclidean is typically used for real-value examples 🡪 important to standardize features so they each contribute equally
  + Some others: Manhattan, Minkowski

# Ch 4: Building Good Training Datasets – Data Preprocessing

* **Topics covered in this chapter:**
  + **Removing and imputing missing values from the dataset**
  + **Getting categorical data into shape for ML algorithms**
  + **Selecting relevant features for the model construction**
* An ML algorithm can at most only be as good as its data
* Most computational tools don’t handle missing values, so it is crucial to take care of them prior to calculation

## Understanding the scikit-learn estimator API

* In the code we used the “SimpleImputer” class 🡪 part of so-called **transformer API** in scikit-learn, which is used for implementing Python classes related to data transformation
  + The 2 essential methods of those estimators are “fit” and “transform”
    - Fit – used to learn the parameters of the training data
    - Transform – uses those parameters to transform the data
* Any data array that is to be transformed needs to have the same # of features as the data that was used to fit the model
* The classifiers in CH 3 used scikit-learns **estimators API** (which is different) 🡪 this one has a .predict() method

## Feature Scaling

* Decision trees and random forest are two of the few ML algorithms that don’t need feature scaling (scale invariant)
* Importance of feature scaling can be illustrated by thinking of two features: one that goes from 1 to 10, a second that goes from 1 to 100,000
  + The squared error function in Adaline (Ch 2) will mostly be busy optimizing the weights according to the large errors in the 2nd feature
  + KNN algorithm with Euclidean distance: the compute distances between examples will be dominated by the 2nd feature
* 2 common approaches to feature scaling: normalization and standardization
  + Normalization – rescaling features to a range of [0, 1], which is a special case of min-max scaling
    - Good when need values in a bounded interval
  + Standardization – can be more practical for ML algorithms that have optimization (gradient descent). The reason is that many linear models such as logistic regression and SVM (Ch 3), initialize weights to 0 or close to 0. Using standardization, we center feature columns at mean 0 with standard deviation of 1 so that the feature columns have the same parameters as a standard normal distribution (zero mean and unit variance), which makes it easer to learn the weights.
    - However, standardization does NOT change the shape of the distribution, and it does NOT transform non-normally distributed data into normally distributed data 🡪 **maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling**

## Selecting Meaningful Features

* If a model performs better on training data than test data, this is a strong indicator of overfitting
  + Overfitting – model has high variance
* Common solutions to reduce generalization error:
  + Collect more training data
  + Introduce a penalty for complexity via regularization
  + Choose a simpler model with fewer parameters
  + Reduce the dimensionality of the data

## L1 and L2 regularization as penalties against model complexity

* L2 regularization (Ch 3) is one approach to reduce the complexity of a model by penalizing large individual weights
  + Defined squared L2 norm of our weight vector, **w**, as follows:
* Another approach to reduce model complexity is the related **L1 regularization**:
  + Here we simply replaced the squared error of the weights with the sum of the absolute values
* In contrast to L2, L1 usually yield sparse feature vectors, and most feature weights will be zero.
  + Sparsity can be useful in practice if we have high-dimensional data with many features that are irrelevant 🡪 In this sense, L1 regularization can be understood as a technique for feature selection
* L1 vs L2 regularization:
  + <https://medium.com/analytics-vidhya/l1-vs-l2-regularization-which-is-better-d01068e6658c>
    - L1 (lasso); L2 (ridge)
    - L1 tries to normalize to median; L2 tries to normalize to mean
    - This explains why L1 can be helpful in feature selection by eliminating unimportant features, it focuses on median rather than outliers that skew the mean
  + <https://www.youtube.com/watch?v=Q81RR3yKn30&ab_channel=StatQuestwithJoshStarmer>
    - The regularization term is really just a way to make your predictions closer to the actual data w/o incorporating new data
    - The large lambda gets the more our predictions become less sensitive to our feature variables
    - Can determine a value of lambda to use by using cross-validation to choose the one with the lowest variance
  + <https://www.youtube.com/watch?v=NGf0voTMlcs&ab_channel=StatQuestwithJoshStarmer>
    - L2 (ridge) is just lease squares + ridge regression penalty
    - As lambda variable increases:
      * Ridge can only shrink slope asymptotically close to zero
      * Lass can shrink slope to zero 🡪 this is how it can exclude useless variables
  + <https://stats.stackexchange.com/questions/495873/lasso-regressions-role-in-shrinking-the-coefficient-to-zero-and-ridge-regressio>
    - Explanation of why lasso can reduce weights to zero

**Sequential Feature Selection Algorithms**

* Dimensionality reduction via feature selection – an alternative way to reduce the complexity of the model and avoid overfitting
  + Especially useful for unregularized models
  + Two main categories of dimensionality reduction techniques:
    - 1) feature selection – select a subset of the original features
    - 2) feature extraction – derive information from the feature set to create a new feature subspace
* Going to look at feature selection 🡪 CH 5 will dive more into feature extraction
* Sequential feature selection algorithms are a family of greedy search algorithms that are used to reduce “d” dimensions to “k” dimension, where k<d
  + Greedy search algorithms make locally optimal choices at each stage of a combinatorial sear problem 🡪 generally yield suboptimal results but require less computation than an exhaustive search algorithm
* **Sequential backward selections (SBS):**
  + Sequentially removes features from the full feature subset until it reaches the desired # of features. To determine which features are removed, need to define a criterion function “J” that we want to minimize
  + Let’s define “J” as performance loss before and after removal, will remove the feature that causes the lease performance loss:
    - 1) initialize algorithm with k=d, where “d” is the dimensionality of the full feature space,
    - 2) determine the feature, , that maximizes the criterion: , where
    - 3) remove the feature, , from the feature set:
    - 4) terminate if “k” equals the desired numbers of features; otherwise, go to step 2

**Ch 5: Compressing Data via Dimensionality Reduction**

**Ch 6: Learning Best Practices for Model Evaluation and Hyperparameter Tuning**

**Ch 7: Combining Different Models for Ensemble Learning**

**Ch 8: Applying Machine Learning to Sentiment Analysis**

**Ch 9: Predicting Continuous Target Variables with Regression Analysis**

**Ch 10: Working with Unlabeled Data – Clustering Analysis**