Week1: The higher the entropy(bits), the lower the purity. Pi - proportion of examples from class i

$$entropy(S) = -\sum_i P_i . \log_2 P_i$$

Normalization(min-max scaling)-Standardization: (Mu-Sigma: mean -standard deviation)

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \qquad x' = \frac{x - \mu(x)}{\sigma(x)}$$

Minkowski distance. q=1:Manhattan(L1 norm), q=2:Euclidean(L2 norm)

$$D(A,B) = (|a_1 - b_1|^q + |a_2 - b_2|^q + \ldots + |a_n - b_n|^q)^{1/q}$$

Similarity between binary vector:
1. Hamming distance = Manhattan for binary vectors
2. Simple Matching Coefficient (SMC) - matching 1-1 and 0-0 / num. attributes
SMC is not suitable for sparse data(contain too many 0 leads to similar)
3. Jaccard coefficient- ignores matching 0-0 both numerator and denominator
Cosine similarity-useful for sparse data(both binary and non-binary)-measure the angle between A and B

$$\cos(A,B) = \frac{A.B}{\|A\|\|B\|}$$

Pearson correlation coefficient - measure linear relationship for variable or vector – range[-1:1]. Also called **"r"**, that is **squared of R^2. This method can't indicate positive or negative.**
-1: perfect negative correlation, +1: perfect positive, 0: no correlation

$$\mathrm{corr}(\mathbf{x},\mathbf{y}) = \frac{\mathrm{covar}(\mathbf{x},\mathbf{y})}{\mathrm{std}(\mathbf{x})\mathrm{std}(\mathbf{y})}$$

where:

$$mean(\mathbf{x}) = \frac{\sum_{k=1}^{n} x_k}{n} \qquad std(\mathbf{x}) = \sqrt{\frac{\sum_{k=1}^{n}(x_k - mean(\mathbf{x}))^2}{n-1}}$$

$$\mathrm{co\,var}(\mathbf{x},\mathbf{y}) = \frac{1}{n-1}\sum_{k=1}^{n}(x_k - mean(x))(y_k - mean(y))$$

Week2: K-Nearest neighbor: Find the nearest training example to it using a **distance measure** that suited for classification and regression. The **class label of the nearest neighbor** will be the predicted label for the new example. Euclidean distance is most frequently used.
Normalization can transform attribute to [0,1], used to avoid the dominance of attributes with large values over attributes with small values when calculating distance. Transform **nominal attributes** to numerical attributes(discretization) before training.
Training is fast because no model is built. **Impractical for large data** due to time and memory requirements.
KNN is **sensitive to value of k**, more nearest neighbors increases the robustness to noisy examples. **Weighted KNN** is a method to improve performance. KNN produces **arbitrarily shaped decision boundary** and **not effective for high-dimensional data**, solution is dimension reduction.
As the number of dimensions increases, all points tend to become equidistant from each other. Increasing computational complexity, and data sparsity.
1R: 1. Generate a rule (decision stump) for each attribute 2. Evaluate each rule on the training data and calculate the number of errors 3. Choose the one with the smallest number of errors.

attri-bute	attribute values & counts	rules	errors	total errors
outlook	sunny: 2 yes, 3 no	sunny -> no	2/5	4/14
	overcast: 4 yes, 0 no	overcast -> yes	0/4	
	rainy: 3 yes, 2 no	rainy -> yes	2/5	

PRISM: 1. Choose a category 2. Define empty rule: if ? then category=True 3. Calculate p/t for each value of each attribute. t->samples have same "value of attribute". P->samples when "category=True" and have same "value of attribute" 4. Add rule with the most p/t 5. Subsample that regulated by rule 6. Repeat 3 and 4 but don't include rule in "each value of each attribute" until get one "value of attribute"' p/t = 1 7. If rule can't include all sample that "category=True", delete sample that regulated by rule and repeat 2-6. After 7 get a new rule that have "or" relationship with previous rule 8. Repeat 1-7 but category=other.
Week3: Linear Regression
Goal: **Prediction** that predict the value of the dependent variable. **Descriptive analysis** that assess the strength of the relationship between x and y

$$y = b_0 + b_1 x$$

intercept slope

The different between prediction and ground true called **prediction error or residual**. Performance index is **sum of squared prediction errors(SSE)**, our goal is select the **line which minimizes SSE** that can be solved using **the method of least squares**.

$$SSE = \sum_i (y_i - \hat{y}_i)^2 \qquad b_1 = \frac{\sum x_i y_i - [(\sum x_i)(\sum y_i)]/n}{\sum x_i^2 - (\sum x_i)^2/n}$$

$$b_0 = \bar{y} - b_1 \bar{x}$$

R^2 **measures the goodness of fit**. **R^2 = SSR/SST** that values between 0 and 1; the higher the better. R^2 value **may be negative** that indicate overfit.
SST = SSR+SSE. **SST**-sum of squared total errors, **SSR**-sum of squared regression errors

$$SSR = \sum_i^n (\hat{y}_i - \bar{y})^2 \qquad SST = \sum_i^n (y_i - \bar{y})^2 = (n-1)\,\mathrm{var}(y)$$

Mean absolute error(MAE)-Mean squared error(MSE)

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \qquad MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Logistic Regression
assumes that the relationship between the feature and class variable **is nonlinear. Logistic Regression is linear regression with sigmoid function**
The equation of logistic (sigmoidal) curve and sigmoid is:

$$p = \frac{e^{b_0 + b_1 x}}{1 + e^{b_0 + b_1 x}} \qquad \sigma(x) = \frac{1}{1 + e^{-x}}$$

p is the probability for class 1. It uses the **maximum likelihood method** to find the parameters b0 and b1- the curve that best fits the data

$$\ln\frac{p}{1-p} = b_0 + b_1 x$$

Compare:
- Logistic regression: $\ln(odds) = b_0 + b_1 x$
- Linear regression: $\hat{y} = b_0 + b_1 x$

The model is **still a linear combination** of the input features, but this combination **determines the log odds of the class** not directly the predicted value.
Overfitting
Small error on the training set but **high error on test set**. The classifier has **not learned to generalize to new examples!**
Reasons: **noise** in training data; **Training data** does not contain enough representative examples – **too small**; Training data very different than test data – **not representative enough; model is too complex** that capture noise and individual data point feature
Underfitting
The **model is too simple** and doesn't capture all important aspects of the data
Regularization
Ridge regression (L2 regularization)
will makes the **magnitude of the coefficients** is as small as possible, i.e. **close to 0**. Minimizes the following cost function:

$$\underbrace{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}_{\text{MSE}} + \underbrace{\alpha\sum_{i=1}^{m} w_i^2}_{\text{regularization}}$$

Increasing Alpha makes the coefficients smaller (close to 0)
Lasso regression
some w will become exactly 0; a form of automatic **feature selection**

$$\underbrace{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}_{\text{MSE}} + \underbrace{\alpha\sum_{i=1}^{m}|w_i|}_{\text{regularization}}$$

Week4: Naive Bayes
Bayes theorem:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Two assumptions: **independence** and **equally importance for all attributes**
How to use: 1. Calculate P(H|E) for each H (class), i.e. P(yes|E) and P(no|E) 2. Compare them and assign E to the class with the highest probability

$$P(yes|E) = \frac{P(E_1|yes)P(E_2|yes)P(E_3|yes)P(E_4|yes)P(yes)}{P(E)}$$

We don't need to calculate P(E), **just compare the numerator**
add 1 to the numerator and **m to the denominator** (m - number of attribute values = 3 for outlook) that is called **Laplace correction or smoothing**. It ensures that the probabilities will never be 0
During **training**, this algorithm **doesn't include the missing values** and just based on training samples without missing values for each attribute
When apply Naive Bayers to **numeric attribute**, using probability density function for a **normal distribution** to calculate probability.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Holdout method
Split the data randomly into 2 sets: training set and test set.
Sometimes we need to use a third set: validation set -> used for hyperparameter tuning; **in ML** we often **just say parameter tuning**, not hyperparameter tuning.
Stratification holdout: Ensures that each class is represented with **approximately equal proportions in both data sets** (training and testing)
Repeated holdout: randomly select a certain proportion for 10 runs. (repeated sample in each runs)
Cross-validation
10-fold cross-validation – typically used
1. Split data into 10 sets set1,.., set10 of approximately equal size
2. A classifier is built 10 times. Each time the testing is on 1 set and the training is on the remaining 9 sets together.
3. Calculate the cross validation accuracy = average (acc1, acc2,...acc10)
Stratified 10-fold cross-validation – this is a standard method
Even better: repeated stratified 10-fold cross-validation
Leave-one-out cross-validation: n-fold cv
Advantage: Makes the best use of data - the greatest possible amount of data is used for training; Deterministic procedure – no random sampling is involved - the same result will be obtained every time
Disadvantage: High computational cost, especially for large datasets
Grid search:
enumerate all parameter combination and evaluate it (using cv or holdout)
Confusion matrix is not performance measure

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Week5: Decision tree
Each **non-leaf node** the corresponds to **a test for the values of an attribute**
Each **branch** corresponds to **an attribute value**
Each **leaf node** assigns a **class**
Top-down learning using **recursive divide-and-conquer** process
Entropy
The **measure of purity(homogeneity)** that we will use is called **information gain**
It is **based on** another measure called **entropy**
In binary tasks, entropy range [0,1], 0 -> all elements of S belong to the same class (max purity, min value of entropy), 1 -> equal number of yes & no
Information gain
measures the reduction in entropy caused by using an attribute to partition the set of training examples
Gain = T1 –T2; **The best attribute is the one with highest information gain**
it will consider the entropies of S1, S2 and S3 and the **proportion of examples** following **each branch (5/14, 4/14, 5/15)**

$$T2 = H(S|outlook) = \frac{5}{14}\cdot H(S_1) + \frac{4}{14}\cdot H(S_2) + \frac{5}{14}\cdot H(S_3)$$

Pruning decision trees
Overfitting: If we grow the decision tree to perfectly classify the training set, the tree may become too specific and overfit the data
Two main strategies
1. **Pre-pruning** - **stop** growing the tree **earlier,** before it reaches the point where it perfectly classifies the training data
2. **Post-pruning** – fully grow the tree, allowing it to perfectly cover the training data, and then prune it
Bottom-up step:
1. Remove the sub-tree rooted at it
2. **Replace it with a leaf with class=majority class of examples** that go along the candidate node
3. **Compare the new tree with the old tree** by calculating the accuracy on the validation set for both
4. **If** the accuracy of the **new tree is better or the same** as the accuracy of the old tree, keep the new tree
Discretizing numeric attributes
Procedure: 1. **Sort the examples** according the values of the numerical attribute. 2. **Split points – whenever the class changes**, halfway. 3. **Evaluate information gain** or other measure **for every possible split point** and **choose the best one**. 4. Information gain for best split point = information gain for the attribute
Gain ratio
highly-branching attributes(like ID code) are more likely to create pure subsets and selected by algorithm
This is **likely to lead to overfitting**
Gain ratio is a modification of information gain that reduces its bias towards highly branching attributes
It **takes into account the number of branches** when choosing an attribute and **penalizes highly-branching attributes**
Benefit for decision trees
Easy to implement
The produced tree is easy to visualize and understand
Interpretability increases the trust in using DT
Variations: purity can be measured in different ways, e.g. CART uses **Gini Index not entropy**
Ensemble
An ensemble **combines the predictions of multiple classifiers**
Ensembles tend to work **better than the base classifiers** they combine
The **base classifiers are independent of each other** that makes ensemble to perform better than a single classifier.
Bagging
Bagging is also called **bootstrap aggregation**
Bootstrap sample D' from D: contains also n **examples**, randomly chosen from D **with replacement**
Especially effective for unstable classifiers (small changes in the training set result in large changes in predictions, e.g. **decision trees, neural networks**)
Step:
Create M bootstrap samples and use each sample to build a classifier
get the predictions of each classifier and combine them with **a majority vote.** i.e. the individual classifiers receive equal weights.
Boosting
using examples that were difficult for the previous classifiers
AdaBoost
Uses a weighted training set
we **increase the weights of the misclassified examples** and **decrease the weights of the correctly classified examples**. There is a mechanism for selecting examples for the training set of the next classifier

D – given training set, m – number of training examples, M – number of models

Model generation

Assign equal weight p_i to each training example *i*, e.g. 1/m
p_i determines the probability of *i* to be selected in the training set for the next model
For k=1 to *M* iterations: //building M models
Create data subset D_k from D with *m* ex. by sampling with replacement using p
Apply learning algorithm to D_k and store resulting model
Compute error e_i of model on each training example *i*:
e_i = 0 if correctly classified, e_i =1 if incorrectly classified
Calculate the weighted error of the model $e = sum(p_i * e_i)$ over all *m* examples
Update the weights:
If example classified correctly by model, multiply its weight by e / (1 - e)
Normalize weights (probabilities) of all examples so that they sum to 1
k=k+1
This results in increasing the weights of the misclassified examples and decreasing the weights of the correctly classified

Classification

For each of the *M* models:
Predict class of testing example using model
Combine predictions using weighed vote, where the weight of each model depends on its accuracy on training set used to build the model

Weak learner is a classifier whose classification performance **is slightly better than random guessing**. **AdaBoost boosts** a set of weak learners into **a strong learner**
Gradient Boosting
Gradient Boosting adds a new model that **minimizes the error of the previous model**
Difference between Bagging and Boosting
Bagging – **separately, equal weighs** to all base learners
Boosting – **iteratively** – the new ones are encouraged to become experts for the misclassified examples by the previous base learners, **different weights**(AdaBoost) based on the performance on training data

- Example for regression tasks, see Géron textbook p.203
- Create model 1: DT1 fit on training data (X,y), store model
- To create model 2:
  - Evaluate DT1 on training data, calculate error:
    y2 = y (actual value) - predicted value by DT1
  - Create model 2: DT2 fit on (X,y2), store model
- To create model 3:
  - Evaluate DT2 on training data, calculate error:
    y3 = y2 - predicted value by DT2
  - Create model 3: DT3 fit on (X,y3), store model
- Now we have 3 decision trees. To make a prediction for a new example: sum the predictions of DT1, DT2 and DT3

Random Forest
using a smaller number of features L (L<K)
n - number of training examples, *m* – number of all features, *k* – number of features to be used by each ensemble member (k<m), *M* – number of ensemble members

Model generation:
For each of *M* iteration
1. Bagging – generate a bootstrap sample
Sample *n* instances with replacement from training data
2. Random feature selection for selecting the best attribute
Grow decision tree without pruning. At each step select the best feature to split on by considering only *k* randomly selected features and calculating information gain

Classification:
Apply the new example to each of the *t* decision trees starting from the root. Assign it to the class corresponding to the leaf. Combine the decisions of the individual trees by majority voting.

Bagging and **random feature selection** are used to **generate diversity** and **reduce the correlation** between the trees.
Ensemble summary
Diversity is generated by manipulating the
1. **training data (Bagging, Boosting)**
2. attributes (**Random Forest = bagging + random selection of attributes**)
3. learning algorithm
Week 6 Support Vector Machines
Support vectors are the examples (data points) that **lie closest to the decision boundary**. The support vectors just **touch the margin** of the decision boundary.
It is **possible to have more than 1 support vector** for each class.
Margin: the **separation between the margin boundary** and the closest examples
The boundary is in the middle of the margin
SVM selects the **maximum margin hyperplane** that has biggest margin because it is likely to have **better generalization** performance

- Minimize $\frac{1}{2}\|\mathbf{w}\|^2$ ← Maximazing the margin
- Subject to the linear constraint $y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1, \forall i$

Another way of expressing correct classification of all training examples i=1..N

- This is an optimization problem that can be solved using Quadratic Programming (QP) and the Lagrange multiplier method
- Firstly, the problem is transformed into an equivalent form using Lagrange multipliers λ :

$$\max \mathbf{w}(\lambda) = \sum_{i=1}^{N}\lambda_i - \frac{1}{2}\sum_{i,j=1}^{N}\lambda_i\lambda_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

Dot product of pairs of training vectors
Class value of the training vectors

$$subject\ to\ \lambda_i \geq 0, \sum_{i=1}^{N}\lambda_i y_i = 0$$

- The values of λs are found using QP
- The solution (i.e. the optimal decision boundary) is given by:

$$\mathbf{w} = \sum_{i=1}^{N}\lambda_i y_i \mathbf{x}_i$$

The **optimal decision boundary** (the maximum margin hyperplane) **w** is a linear combination **(coefficient Lambda * target value*training vector) of the training examples**
The **training examples xi with non-zero Lambda(i)** are the support vectors

To classify a new example z:

$$f = \mathbf{w}\cdot\mathbf{z} + b = \sum_{i=1}^{N}\lambda_i y_i (\mathbf{x}_i \cdot \mathbf{z}) + b$$

Dot product of the new vector and the support vectors

$$sign(f)$$

i.e. the new example belongs to class 1, if f>0 or class -1 if f<0
Soft-margin
C is a hyper-parameter that allows for a **trade-off** between **maximizing the margin** and **minimizing the training error**
Large C: more **emphasis on minimizing the training error**
Non-linear SVM
Transform the data from its original feature space **to a new space where a linear boundary can be used to separate the data**
If the transformation is **non-linear** and **to a higher dimensional space**, it is more likely than a linear decision boundary can be found in it
The **learned linear decision boundary** in the new feature space **is mapped back to the original feature space**, **resulting in a non-linear decision boundary** in the original space
Kernel trick
Method for computing the dot product of a pair of vectors in the new space **without first computing the transformation of each vector from the original to the new space**

This relationship is specified by the kernel function: $\Phi(\mathbf{u})\cdot\Phi(\mathbf{v}) = (\mathbf{u}\cdot\mathbf{v})^2$

$$K(\mathbf{u},\mathbf{v}) = \Phi(\mathbf{u})\cdot\Phi(\mathbf{v}) = (\mathbf{u}\cdot\mathbf{v})^2$$

Functions K for which this is true (i.e. such phi exist) need to satisfy **Mercer's Theorem**

Some kernels satisfying the Mercer's Theorem are

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}\cdot\mathbf{y}+1)^p - polynomial\ kernel$$

$$K(\mathbf{x},\mathbf{y}) = e^{-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}} - RBF$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(k\mathbf{x}\cdot\mathbf{y} - \theta) - tanget\ hyperbolic$$

$$(satisfies\ Mercer's\ Th. only\ for\ some\ k\ and\ \theta)$$

- Training:

$$\max \mathbf{w}(\lambda) = \sum_{i=1}^{N}\lambda_i - \frac{1}{2}\sum_{i,j=1}^{N}\lambda_i\lambda_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

$$subject\ to\ \lambda_i \geq 0, \sum_{i=1}^{N}\lambda_i y_i = 0$$

- Optimal hyperplane in the new space: $\mathbf{w} = \sum_{i=1}^{N}\lambda_i y_i \Phi(\mathbf{x}_i)$
- Classifying new example z:

$$f = \mathbf{w}\cdot\mathbf{z} + b = \sum_{i=1}^{N}\lambda_i y_i K(\mathbf{x}_i, \mathbf{z}) + b$$

PCA
Problems with high dimensional data:
1. **Slower training;**
2. Unreliable classification – examples are **far away from each other**; high dimensional data is very **sparse;**
3. **Overfitting is more likely** in high dimensional data;
4. Building **interpretable** models is **not possible;**
PCA is often called a **feature projection method**
The **new axes capture the essence of the data** (the variability of the data)
It is an **unsupervised method** – it doesn't use the class information
It **projects the data into a lower dimensional space** that **still captures the important information**
The **resulting dataset** (the projection) can **be used as an input** to train a ML algorithm. PCA can also **be used for compression**
Main idea
Given: *N* examples with dimensionality *m* (i.e. *m* features)
Find: *m* new axes $Z_1, \ldots, Z_m$ orthogonal to each other such that
$Var(Z_1) > Var(Z_2) \ldots > Var(Z_m)$
Z1,…, Zm are called principal components

They are **ordered based on** how much **variance** they capture
How to select principal components
Method 1: Set min % of variance that should be preserved, e.g. 95%
- Choose *k* such that $Z_1, Z_2, \ldots Z_k$ capture 95% of the variance

Method 2: (Elbow method)
- Plot the number of dimensions as a function of variance
- There is usually an elbow in the curve where the variance stops growing fast

This is an application of PCA for **feature extraction** – **find a lower dimensional representation** that is **better suited than the original representation**
Singular Value Decomposition (SVD)
Theorem: Any *n* x *m* matrix X (*n* ≥*m*) can be written as the product of 3 matrices

$$X = U \times \Lambda \times V^T$$

U - *n* x *m* orthogonal matrix
VT - the transpose of an *m* x *m* orthogonal matrix
Λ - *m* x *m* diagonal matrix containing the singular values (positive or zero elements)
V defines the new set of axes (principal components)
- Provides important information about the variance in data – the 1st axis goes in the direction with highest variance, 2nd – 2nd highest variance and so on

X is the original data
U is the transformed data, i.e. the *i*-th row of **U** contains the coordinates of the *i*-th row of **X** in the new coordinate system

Compression ratio
Compression ratio = after compression/ before compression

$$r = \frac{k(1+n+m)}{n \times m}$$

For *n* >> *m* >*k*, this ratio is approximately *k/m*
e.g. if *m* = 365 and *k* = 10 => *r* = 0.027 = 2.7%
Week7 Neural Network
Neural networks **consist of neurons** (units, nodes) that connected with each other with directed links where each **connection** has an associated **numerical weight**
The neurons are typically organized into layers: Input layer, output layer and 1 or more hidden layers
Perceptron
The **simplest** neural network is called perceptron
The **stopping condition** is checked at the **end of each epoch**

Perceptron for one epoch

Update weights for each sample

Epoch=1, w=[0.3 0.2 0.4], b=0.1

Ex.	input	output
1	1 0 0	0
2	1 0 1	1
3	1 1 0	0

Apply Ex.1: [1 0 0], t=0

a=step([0.3 0.2 0.4][1 0 0]+0.1)=step(0.4)=1, incorrect; e=t-a=0-1=-1

w_new=[0.3 0.2 0.4] + (-1)[1 0 0]=[-0.7 0.2 0.4]

b_new=0.1+(-1)=-0.9

Apply Ex.2: [0 1 1], t=1

a=step([-0.7 0.2 0.4][1 0 1]-0.9)=step(-1.2)=0, incorrect; e=t-a=1-0=1

w_new=[-0.7 0.2 0.4]+ (1)[1 0 1]=[0.3 0.2 1.4]

b_new=-0.9+1 =0.1

Apply Ex.3: [1 1 0], t=0

a=step([0.3 0.2 1.4][1 1 0]+0.1)=step(0.6)=1, incorrect; e=t-a=0-1=-1

w_new=[0.3 0.2 1.4]+ (-1)[1 1 0]=[-0.7 -0.8 1.4]

b_new=0.1-1 =-0.9

If the training examples are **linearly separable**, the **perceptron** learning rule **is guaranteed to converge** to a solution

A **2-layer** perceptron can solve the **XOR problem**!

Multi-layer perceptron

The transfer function needs to be **differentiable**

Most widely used transfer function: **sigmoid**

Backpropagation

By defining an **error function** and using the **gradient descent** algorithm to calculate Weight Change

The step that is used to move downhill is called **learning rate**

The **gradient descent** algorithm **is not guaranteed to find the global minimum**

The **standard gradient descent** algorithm **sums** the error of **all training** examples and then updates the weights

The Stochastic Gradient Descent **(SGD)** updates the weights after **each example**

Mini-**batch SGD** – **sum** the error of **mini-batches** of training examples

Universal approximators – theorems:

1. **Any continuous function** can be approximated with arbitrary small error by a network with **one hidden layer**
2. **Any function (including discontinuous)** can be approximated to arbitrary small error by a network with **two hidden layers**

Design issues – architecture

Number of hidden layers and neurons: Too many – overfitting, Too few – underfitting

Learning rate

- Too small – **slow convergence**
- Too big – **oscillation, overshooting of the minimum**
- It is not possible to determine the best learning rate before training as it changes and depends on the error surface

Alternatives: variable learning rate – time-dependent

Start with a high learning rate, gradually decrease it – motivation:

- **Big** learning rate initially -> greater weight change - can **reduce** the number of **training epochs** and may even **help to jump over some local minima**
- Later - **decrease** the learning rate to **prevent overshooting the global minimum**

Momentum

Make the current update dependent on the previous by introducing a hyperparameter called momentum

Reduces oscillations and allows to use a larger learning rate

Weights initialization

The **performance** is **very sensitive to the weights initialization** (w and b)

- Random (standard approach) - small random values, e.g. $\mathbb{R}$ [-1, 1]
- Xavier initialization - the weights are sampled from normal distribution

Vanishing gradient problem

Sigmoids saturate and kill gradients. A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. (same to Tanh)

The **weight changes** for the **lower levels are very small**; these layers learn slower than the higher hidden layers

The **derivative** of sigmoid function is **close to 0** when x is too large or too small

Dropout

A method for preventing overfitting

Dropout **forces the NN to be less dependent on certain neurons**, to collect more evidence from other neurons => to be more robust to noise

During training, at each iteration of the backpropagation, we **select randomly neurons in each layer and set their values to 0**

SoftMax:

Converting output values to probabilities

Cross entropy loss(CCE loss)

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad L = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

Week 8 Convolutional Neural Networks

Traditional NNs (multilayer feedforward networks) do not consider the **spatial structure of the image**

CNNs do this by using new types of layers - **convolutional** and **pooling**

Traditional NNs (multilayer feedforward networks) are sensitive to the **position of an object in the image**; CNN are more **robust to shifts/translation**

Irena Koprinska, i

The **filters** convolve with the image parts and **"activate" (have high values) when a specific feature is detected**

Values of filters are learned during training

Pooling

"Summarizes" the input from the previous layer

Reduces the number of **parameters** and helps **prevent overfitting**

Improves robustness to **shifts** in the receptive field

Convolutional dimension function:

$$H_{out} = \left\lfloor \frac{H_{in} - K + 2P}{S} \right\rfloor + 1 \quad W_{out} = \left\lfloor \frac{W_{in} - K + 2P}{S} \right\rfloor + 1$$

Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) are used for **sequential data**

Simple RNN (also called **Elman network**)

Hidden state and output function:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \qquad y_t = W_{hy}h_t + b_y$$

RNN are susceptible to the **vanishing gradient problem**

Reason: during the backward pass, the error gradients will be multiplied by the Whh matrix multiple times - once for each time-step. This repeated multiplication may cause the gradient to vanish if the weights are too small

LSTM

LSTMs introduce a way to maintain a **"long-term memory"** and thereby **mitigate the vanishing gradient problem**.

Forget gate layer: Decides what information from the long-term state Ci-1 should be thrown away or kept

Input gate layer: Decides what new information should be stored in cell state.

Update the cell state: Update the old cell state $Ct-1$ to produce the new state Ct by combining the previous 2 results (the values of the forget and input layers)

Output gate layer: Decides what exactly to output - the output will be based on the cell state, but it will be a filtered version

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$\tilde{c}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot \tanh(C_t)$$

LSTM is **computationally expensive**, it has **4 times more parameters than RNN**.

Week 9 Clustering

The process of **dividing the data objects into groups** (called clusters);

similar to each other **within the cluster**; **dissimilar** to the objects **in other clusters**

Clustering is unsupervised learning: no labels

Similarity between 2 data points A and B is measured **using a distance measure**

Centroid C – the **"middle" of the cluster** - Typically **not an actual data** point in the cluster

Medoid M – the **centrally located data** point in the cluster

Distance between clusters

Single link (MIN) – The **smallest pairwise distance** between elements **from each cluster**

Complete link (MAX) – the **largest** pairwise distance between elements from each cluster

Average link – average pairwise distance between elements from each cluster

K-Means

Partitioned clustering algorithm

1: Select K points as the initial centroids.
2: **repeat**
3: Form K clusters by assigning all points to the closest centroid.
4: Recompute the centroid of each cluster.
5: **until** The centroids don't change

The algorithm is **sensitive to the initial centroids** – different clusters will be produced

Selecting good initial centroids

Method 1: **Run** K-means **several times with different randomly** selected **initial centroids**; evaluate each clustering using SSE, **select** the clustering with the **smallest Sum of Squared Error (SSE)**

SSE: For each point, the **error is the distance to the closest centroid**

Method 2: K-means++

K-means++ is a variation of K-means which uses **a new method for selecting the initial centroids**

Select step:

Select centroids incrementally, until k centroids have been selected

At every step, each point has a probability to be selected as a centroid that is proportional to the square of its distance to its closest centroid

=> Selects points that are farthest away from the current centroids – selects well-separated points

Can select outliers, but outliers are rare by definition

Handling empty clusters

K-means can yield empty clusters – no points are allocated to a cluster during the assignment step – the cluster consists only of the initial centroid

Solution: choose different initial centroid – strategies:

- Choose the point that is farthest away from any current centroid
- Use the K-means++ approach (similar idea)
- Choose a point from the cluster that has the highest SSE
  - This will typically split the cluster and reduce the overall SSE of the clustering

K-means issues: Outliers

When there are outliers, the **resulting cluster centroids** are **less representative** and the **SSE is higher**

Solution: **remove outliers before clustering**

Variations: Bisecting K-means

Main idea: to obtain k clusters, **split the data into 2 clusters, select one of these clusters to split further** and so one until k clusters are obtained

Conclusion K-means

Doesn't work well for natural clusters with complex **(non-spherical)** shape/ **different sizes/ different density**

bisecting K-means and K-means++: Both **reduce sensitivity to initialization** (choice of initial centroids)

Gaussian Mixture Model (GMM)

is a **probabilistic** clustering

Each component of GMM has parameters of mean, covariance, weight.

It **assumes** that the **data is generated by a mixture of K normal (Gaussian) distributions. One distribution corresponds to 1 cluster**

After each estimation, we **compute the probability of each example** to **belong to each distribution** (cluster)

Algorithm 9.2 EM algorithm.

1: Select an initial set of model parameters.
(As with K-means, this can be done randomly or in a variety of ways.)
2: **repeat**
3: **Expectation Step** For each object, calculate the probability that each object belongs to each distribution, i.e., calculate $prob(distribution\ j|\mathbf{x}_i, \Theta)$.
4: **Maximization Step** Given the probabilities from the expectation step, find the new estimates of the parameters that maximize the expected likelihood.
5: **until** The parameters do not change.
(Alternatively, stop if the change in the parameters is below a specified threshold.)

GMM can be seen as a **generalization of K-means** – it **allows for elliptical clusters** rather than circular and for **probabilistic assignment** to each cluster rather than crisp

Hierarchical clustering

Produces a set of **nested clusters** organized as a hierarchical tree

Can be visualized as a **dendrogram.** The dendrogram provides a useful visualization – an **interpretable description** of the clustering process

Computationally expensive which limits its applicability to high dimensional data. **Sensitive to noise and outliers**

Outliers are more **problematic for the Ward's method** as they increase the SSE and **less problematic for the single, complete and average link**

Two approaches

Agglomerative (bottom-up)

The key operation is computing the **distance between two clusters** (the **proximity matrix**)

1. Compute the proximity matrix
2. Let each data point be a cluster
3. **Repeat**
4. Merge the two closest clusters
5. Update the proximity matrix
6. **Until** only a single cluster remains

Hierarchical clustering typically uses the following **distance measures**:

- Single link (MIN)
- Complete link (MAX)
- Average link
- **Ward's method** – the distance between 2 clusters is the increase in SSE that results when the 2 clusters are merged

Divisive (top-down)

Minimum spanning tree: Each time, select the point with the smallest distance to the tree and add it to the tree (create a link)

Split: Each time, disconnect the edge with the longest distance (lowest similarity)

Algorithm 7.5 MST Divisive Hierarchical Clustering Algorithm

1: Compute a minimum spanning tree for the proximity graph.
2: **repeat**
3: Create a new cluster by breaking the link corresponding to the largest distance (smallest similarity).

Minimum spanning tree

Week 10 Density-based: DBSCAN

DBSCAN – **D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise

In contrast to K-Means which finds circular clusters, DBSCAN can find clusters with **arbitrary and complex shape**

Neighborhood of a point A = the area within a radius **Eps**

Density of a point A = the **number of points** in the neighborhood of A, **including** the point A

Density threshold **MinPts** – the **minimum number of points in the Eps** neighborhood of a point

The density of a point depends on the neighborhood Eps

Eps **too big** - all points will have a density of m (m =number of points in the dataset) – **all points will be core point**

Eps **is too small** – all points will have a density of 1 – **all points will be noise**

3 types of points

Core point – if its Eps neighborhood contains at least MinPts points.

Border point – if it is not a core point but falls within the neighborhood of a core point.

Noise point – if it is not a core or a border point

Algorithm

Algorithm 8.4 DBSCAN algorithm.

1: Label all points as core, border, or noise points.
2: Eliminate noise points.
3: Put an edge between all core points that are within Eps of each other.
4: Make each group of connected core points into a separate cluster.
5: Assign each border point to one of the clusters of its associated core points.

Ties need to be resolved when a **border point belongs to** the neighborhood of **more than 1 core point**

Method for selecting Eps and MinPts

Examine the distance from a point to its k-th nearest neighbor (**k-dist** –> k generally = minpts-1)

For **points** that **belong to some cluster, k-dist will be small** (if k is not larger than the cluster size)

After calculating k-dist for all points, sort all points (from small to large). The **horizontal** axis of the icon is the **number of the point**. After finding the **elbow**, the **k-dist near the elbow can be selected as eps**. The **area before the elbow is dense**, and the **area after it jumps to sparse area**.

Time and space complexity

Time – O(n2), n - number of points

Space – O(n)

Strengths

Can form clusters of **arbitrary shapes and different sizes** => can find clusters that K-means cannot

Does **not require the number of clusters to be pre-specified** (but this is done indirectly by specifying Eps and MinPts)

Resistant to noise as it uses a density-based definition of a cluster and labels points as noise

Weaknesses

DBSCAN cannot handle well clusters with **widely varying density**

Does not work well for **high dimensional data** – more difficult to define density

Sensitive to the input parameters **Eps and MinPts**

Eps and MinPts may be **difficult to determine**

Grid-based clustering

Split the values of each attribute into intervals, creating grid cells

the **density** is the number of points per amount of space

Forming clusters from adjacent groups of dense cells is straightforward

It is an efficient algorithm: **a single pass through the data can determine the grid cell** of every object and the count of every cell

Different clusters may exist in different dimensions (subsets of attributes)

CLIQUE algorithm

Provides an efficient technique for **searching for clusters in subspaces**

CLUQUE (**CL**ustering **I**n **QUE**st) – dimension-growth, **subspace grid-based clustering**

It finds the subspace of the highest dimensionality where high-density clusters exist

Apriori principle (alco called monotonicity principle): If a set of points forms a density-based cluster in k-dim space, then the same set of points is also part of a density-based cluster in all possible subsets of those dimensions

Evaluating clustering results

Unsupervised Method 1: Measuring cohesion and separation

A good clustering produces clusters with

- **high cohesion** = high similarity within the cluster
- **high separation** = low similarity between the clusters

Cohesion of a cluster K_i:

$$cohesion(K_i) = \sum_{x \in K_i} dist(\mathrm{x}, c_i)$$

c_i is the cluster centroid, x are all the objects in this cluster

Cohesion of the **whole clustering** - overall cohesion over all clusters Ki:

cohesion =cohesion(K1)+cohesion(K2)=2

Separation of a cluster K_i from the other clusters:

$$separation(K_i) = dist(c_i, c)$$

c is the overall centroid (all points considered)

$\mathbf{c_i}$ are the cluster centroids, i=1,2

- Separation of the clustering - overall separation weighted by the size of each cluster |K_i|:

$$separation = \sum_{i=1}^{k} |K_i| dist(c_i, c)$$

- Sum of the separations of the individual clusters, weighed by their size

We can use **squared distances**(like SSE) to express cohesion and separation

Silhouette coefficient $s_i = (b_i - a_i)/\max(a_i, b_i)$

ai = the average distance from i to all points in its cluster (cohesion)

bi = the average distance from i to all points in another cluster (separation)

Values between -1 and 1; the **higher the better**

Unsupervised Method 2: Correlation between 2 similarity matrices

Compare the similarity matrix of different clustering results. It's suitable for **continuous date**.

Flatten the two matrices and use **Pearson correlation function** to compute coefficient

Unsupervised Method 3: Visual inspection of sorted similarity matrix

Supervised Method 1: Entropy and purity

Entropy of a cluster K: $entropy(K) = -\sum_i P_i \log_2 P_i$

- i – index over the class labels (ground truth), e.g. banana and apples
- P_i - proportion of examples that belong to class i, base of log is typically 2

The smaller the entropy, the greater the purity of the cluster

weighted by the size of each cluster

$$overall_entropy = \sum_{i=1}^{k} \frac{m_i}{m} entropy(K_i)$$

Num. of examples in each cluster/total number of examples

Purity is similar to entropy

$$purity(K) = \max_i Pi \qquad overall_purity = \sum_{i=1}^{k} \frac{m_i}{m} purity(K_i)$$

Pi refers to **maximum number of true lable/total number**

Supervised Method 2: Correlation between obtained and ideal similarity matrix

Use **Simple Matching Coefficient** (SMC) (also called Rand statistic) or the **Jaccard coefficient** to calculate the similarity between matrix

It's suitable for **binary** data.

Determining the Number of Clusters

Elbow method

- Run the clustering algorithm for several k, plot SSE or other unsupervised measure vs number of clusters

Clustering Tendency

Generate 2 sets of *p* points (*p*<<*m*, heuristic: *p*=0.1**m*):

- Set 1: *p* random points
- Set 2: *p* actual points from the dataset - sample randomly *p* points of the given *m* points

For each point in **set 2**, calculate its nearest neighbor distance to **other points in the original data set**, denoted as u_i.

For each point in **set 1**, calculate its nearest neighbor distance to **all points** in the original data set, denoted as w_i.

Near 1: data is highly clustered

Near 0.5 => spatially random data

Hopkins statistic H:

$$H = \frac{\sum_{i=1}^{p} u_i}{\sum_{i=1}^{p} u_i + \sum_{i=1}^{p} w_i}$$

Week 11 Markov Models

A Markov chain (also called Markov process) in which the **probability** of each state **depends only on the previous state**

Hidden Markov models

that allow us to **predict a sequence of hidden events** from a set of **observed events**

HMM problem 1 (Evaluation)

Question: **how likely is this sequence?**

Solution by enumeration - compute the probability of each possible state sequence and sum these probabilities

This method will be **slow** for tasks with long sequences and many states!

Forward algorithm:

Step 1: Initialization: $f_k(1) = A_0(k)e_k(x_1)$

- $f_k(1)$ - forward probability of state *k* at time *1*
- x_1 = *Shirt*
- 3 states: *Rainy, Cloudy, Sunny*

$f_{Rainy}(1) = A_0(Rainy)\ e_{Rainy}(Shirt)$ = 0.6*0.8=0.48

$f_{Cloudy}(1) = A_0(Cloudy)\ e_{Cloudy}(Shirt)$ = 0.3*0.5=0.15

$f_{Sunny}(1) = A_0(Sunny)\ e_{Sunny}(Shirt)$ = 0.1*0.01=0.001

Step 2: Iteration

$f_k(i)$ - forward probability of state *k* at time *i*: $f_k(i) = e_k(x_i)\sum_j f_j(i-1)\,a_{jk}$

- time step *t* = 2, x_2 = *Hoodie*
- 3 states: *Rainy, Cloudy, Sunny*

1) For *Rainy*:

$f_{Rainy}(2) =$

$= e_{Rainy}(Hoodie)\left(f_{Rainy}(1)a_{RainyRainy} + f_{Cloudy}(1)a_{CloudyRainy} + f_{Sunny}(1)a_{SunnyRainy}\right)$

= 0.01*(0.48*0.6+0.15*0.4+0.001*0.1)=0.01*0.3481=0.0035

Step 3: Termination

- $f_k(m)$ - forward probability of state *k* at time *m* – at the last time step
- *m* = 2

$$P(X) = \sum_k f_k(m)$$

$P(X) = P(Shirt, Hoodie) = f_{Rainy}(2) + f_{Cloudy}(2) + f_{Sunny}(2) =$

$= 0.0035 + 0.0189 + 0.0739 = 0.0963$

HMM problem 2 (Decoding)

what is the most likely sequence of hidden states?

Viterbi algorithm

Step 1: Initialization: $V_k(1) = A_0(k)e_k(x_1)$

- $V_k(1)$ - Viterbi score of state *k* at time *1*
- x_1 = *Shirt*
- 3 states: *Rainy. Cloudy. Sunny*

$V_{Rainy}(1) = A_0(Rainy)\ e_{Rainy}(Shirt)$ = 0.6*0.8=0.48

Step 2: Iteration

- $V_k(i)$ - Viterbi score of state *k* at time *i*
  $V_k(i) = e_k(x_i)\max_j V_j(i-1)a_{jk}$
- time step *t* = 2, x_2 = *Hoodie*

1) For *Rainy*

$V_{Rainy}(2) = e_{Rainy}(Hoodie)$ *

$\max\left(V_{Rainy}(1)a_{RainyRainy},\ V_{Cloudy}(1)a_{CloudyRainy},\ V_{Sunny}(1)a_{SunnyRainy}\right)$

= 0.01*max(**0.48*0.6**, 0.15*0.4, 0.001*0.1)=0.01*0.48*0.6=0.0029

$Ptr_{Rainy}(2) = \mathrm{argmax}(0.48*0.6, 0.15*0.4, 0.001*0.1) = 1$, i.e. Rainy

Step 3: Termination and trace-back

$\pi_N = \mathrm{argmax}_k V_k(N)$

The final state (it is for time t=2) is given by:

$\mathrm{argmax}\left(V_{Rainy}(2), V_{Cloudy}(2), V_{Sunny}(2)\right) = \mathrm{argmax}\left(V_{Rainy}(2), V_{Cloudy}(2), V_{Sunny}(2)\right) =$

$= \mathrm{argmax}(0.0029, 0.0144, 0.0379) = 3$, i.e. Sunny

Backtracking through the pointers:

$Ptr_{Sunny}(2) =$ Rainy

The largest one here is Vsunny(2). Next, find which Vx(1) is the largest in Vsunny(2). Then x is the previous hidden state, and so on.

HMM problem 3 (Learning)

What is the model Lambda = (Pi, *A*, *A*0)?

1. Initialize $\lambda = (\pi, A, A_0)$ to random values
2. Compute the probabilities A, and A_0 for all time steps, for all states π (expectation step)
3. Re-estimate $\lambda = (\pi, A, A_0)$ based on the observation sequence X – use the probabilities to predict the most likely observation sequence and compare with the actual observation sequence (maximization step)

If P(X| λ) have increased, return to step 2, otherwise stop.

(2) **Forward** and **backward probability**, at time step t, the probability of observing the previous sequence and observing the subsequent sequence

(3) According to the forward and backward probability, the initial, transition and emission probability are **recalculated according to the formula**

Week 12 Reinforcement Learning

Learn to make good sequence of decisions

The purpose of RL: **Maximize the expected sum of rewards!**

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: **transition probability**
i.e. distribution over next state given (state, action) pair

Policy: *agent's behaviour function*
Value Function: *how good is each state and/or action*
Model: *agent's representation of the environment*

γ : discount factor

Policy: The purpose of MDP(Markov Decision Process) is to find a policy that maximizes the expected value of the cumulative reward. The policy is the probability distribution of taking action a in state s.

Value function: The reward obtained by performing action a in state s

State-Value Function (How good is a state?): The value function at state s, is the **expected cumulative reward** from **following the policy from state s**

Action-Value Function (How good is a state-action pair?): The **Q-value** function at state s and action a, is the **expected cumulative reward** from **taking action a in state s** and **then following the policy**

Bellman equation

Q^* *satisfies the following* ***Bellman equation***, i.e., immediate reward plus discounted value of successor state

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

It represents the **maximum expected cumulative reward** that can be obtained after performing action a in state s and then choosing the "**optimal policy**".

This equation recursively calculates the action-value function by choosing the "optimal action" in each subsequent state.

Q-Learning

By **updating the Q value** (the value of the Q function), the Q function **is made close to the optimal Q value function;**

In the end, the **optimal strategy** is the action a with the **maximum Q value in each state s**

Deep Q Learning

In **traditional Q-learning**, the Q-value function is stored in the form of **a lookup table**, which is feasible when the state and action space is small, but becomes **impractical in high-dimensional space**.

Deep Q-learning solves this problem by **using a neural network** to approximate the Q-value function.

Output are state-action values from all possible actions

Divergence issues using neural networks

– Correlations between samples

– Non-stationary targets

Solution: **Experience replay**

1. Build data set from agent's **own experience**
2. **Sample experiences uniformly**(randomly sample mini-batch) from data set to **remove correlations**

To deal with **non-stationarity**, target parameters **w** are held **fixed**

Double DQN

Q-learning is known to **overestimate state-action values(Q-value)** -> The max operator uses the **same values** to **select and evaluate** an action

The upward bias can be removed by **decoupling the selection from the evaluation**

The difference from DQN is that calculating the target function Q value:

DQN directly **uses the target network** to calculate

Double DQN first uses the **main network to select actions** and **then uses the target network to evaluate** (calculate the target Q value)

Improvements: Prioritized Replay

Assign **weight experiences** according to their significance

Use **stochastic sampling** to increase sample diversity

DL is a general-purpose framework for **representation learning**

Deep reinforcement learning: AI = RL + DL