ml with pytorch book

some syntactical sugar while learning from ml book
(≧◡≦) ♡

chapter 1 : giving computers ability to learn from data

fundamentals, types machine learnings, structure of training data, workflow of training

chapter 2 : training simple ml algorithms for classification

`page no. 19`

- perceptron and adaline network
- gradient descent
- full batch and stochastic gradient descent
- using python oop approach to perform fit method (training method)
- feature scaling (ensuring they are on similar scale) and standardization
- learning rate and its importance
- parameters - hyperparameters,

chapter 3 : ml classifiers using sickit-learn

`page no. 53 to 105`

jupyter notebook with all code exercises

- training perceptron using scikit-learn
- logistic regression
    - in logistic regression we have formulae for **maximum likelihood** and we try to maximize likelihood of observing given data while weights are given
    - but by convention we minimize the loss function instead of maximizing likelihood so we convert our mle (maximum likelihood estimession) into loss function which is also called as **log loss** or **binary cross entropy loss**
    - $L(w) = -\ell(w)$, here first term is loss function and second is mle
- converting adaline's oop approach for logistic regression
- gradient descent algorithm for logistic regression
- training logistic regression using scikit-learn

overfitting vs regularization

- overfitting is when model learns too well on training data but sucks for unseen data
- its caused due to high number of parameters and high variance
- under fitting is when we have less parameters or our model isn't complex enough so it does not learn patterns in data and does random guesses
- bias-variance tradeoff
    - high variance is is proportional to overfitting
    - high bias is proportional to underfitting
    - bias is like error or how far are we from real values in general
    - and variance is like how consistently our model learns on each data along with patterns it trains on noise of data like random examples or outliers and hence its general prediction is bit wrong
    - to find good tradeoff between bias and variance we use regularization
- regularization:
    - we add extra term to penalize the extreme weight values
    - formulae (for L2 regularization)

$$\frac{\lambda}{2n}\|w\|^2 = \frac{\lambda}{2n}\sum_{j=1}^{m} w_j^2$$

    - we add regularization term to loss so that our loss has extra information and our model doesn't train on
    - for logistic regression we add this regularization term in loss function as given

$$L(w,b) = \frac{1}{n}\sum_{i=1}^{n}\left[-y^{(i)}\log(\sigma(z^{(i)})) - (1-y^{(i)})\log(1-\sigma(z^{(i)}))\right] + \frac{\lambda}{2n}\|w\|^2$$

    - gradient of loss without regularization term

$$\frac{\partial L(w,b)}{\partial w_j} = \left(\frac{1}{n}\sum_{i=1}^{n}(\sigma(w^T x^{(i)}) - y^{(i)})x_j^{(i)}\right)$$

    - gradient of loss with regularization term

$$\frac{\partial L(w,b)}{\partial w_j} = \left(\frac{1}{n}\sum_{i=1}^{n}(\sigma(w^T x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{n}w_j$$

    - in sklearn we have term c as parameter in logisticregression function this c is inversely proportional to our regularization parameter γ
    - by increasing regularization strength we can reduce the risk of overfitting but having too big of regularization strength can cause underfitting due to

support vector machines

detailed resource by andrew ng

- its like perceptron but in perceptron we minimize the loss and in svm we maxmize margin
- **margin** :
    - its the distance between decision boundary (hyperplane) and support vectors (training examples that are closest to decision boundary)
    - A larger margin implies better generalization to unseen data
    - models with small margin can be prone to errors and overfitting
- In a dd-dimensional space, a **hyperplane** is a flat affine subspace of dimension d−1
- we use **slack variable** so we can separate non-linearly separable data
- use slack variable adds another variable called **C**
- C is hyper parameter for controlling penalty for misclassification, large values of C gives large error penalties and smaller values of C correspond to less penalty.
- **support vectors** -> datapoints closest to decision boundary
- big c -> small γ -> small extra term to add regularization -> model trains too good on data -> small margin and overfitting
- small C -> big γ -> big extra term to add regularization -> model generalises instead of memorizing data -> big margin , general boundary

> linear svm and linear logistic regression are similar but logistic regression is more prone to overfitting as it cares about maximizing the likelihood of data. but logistic regression is also easier to implement math behind is simpler and it can be easily updated which is crucial when you have streaming data.

- sklearn uses LIBLINEAR and LIBSVM libraries to implement this svm for faster calculations
- we can use alternative method of using SGDClassifier class instead of SVC when we have huge amount of data, SGDClassifier works similar to stochastic gradient algorithm
- **kernel methods:**
    - when we have linearly non separable data we transform our data into higher dimensions so it becomes linearly separable
    - we use mapping function for ϕ to transform out data in to high dimensions space

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

    - when we have higher dimension space we saperate data with hyperplace because data in that space is separable by linear decision boundary and we then project it to lower dimension space the decision boundary gets converted into non-linear shape

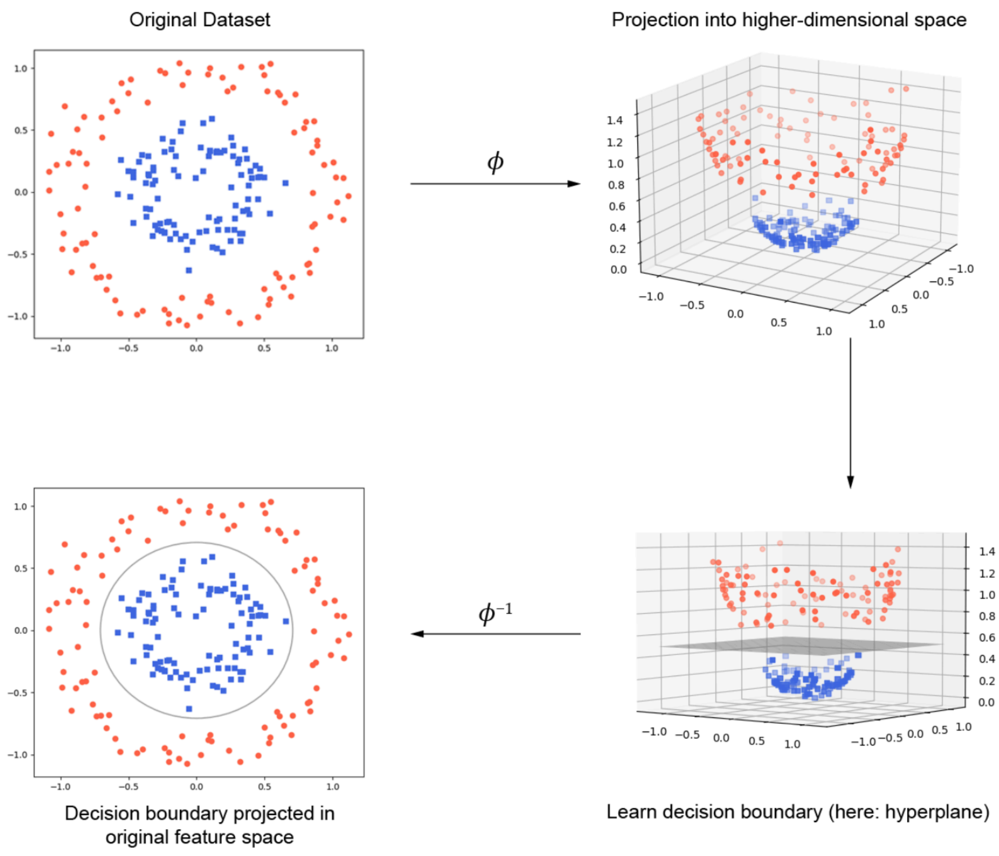


Figure 3.14: The process of classifying nonlinear data using kernel methods

- **kernel trick to find separating hyperplanes :**
    - we can use mapping function ϕ to transform data into higher dimensional space and then train our model to classify data in high dimensional space and whenever new data comes we can transform it using mapping function but this process is computationally very expensive
    - kernel trick is where we use **kernel function** to calculate datapoints in high dimensions without actually calculating them
        - if mapping function $\phi(x)$ is the mapping function for x then kernel function for $K(xi, xj)$ is dot product between transformed datapoints for example

$$K(xi, xj) = \phi(xi) \cdot \phi(xj)$$

    - **How the Kernel Trick Works in SVMs**
        1. The SVM optimization problem involves computing dot products between data points (e.g., $x_i \cdot x_j$).
        2. Instead of computing these dot products in the original space, the kernel function computes them in the transformed high-dimensional space:

$$\phi(x_i) \cdot \phi(x_j) = K(x_i, x_j)$$

        3. This allows the SVM to find a decision boundary in the high-dimensional space without explicitly transforming the data.
    - we have various kernel functions but in svm we use **radial basis function (RBF)** kernel also called **Gaussian kernel**. it looks like

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$$

- where $\gamma = \frac{1}{2\sigma^2}$ and σ = spread
- this kernel function help is finding similarity between two datapoints
- as value of gamma increases -> spread less -> more tight boundary -> can cause overfitting

decision trees

- in decision trees our model learn to predict correct questions (technically called **splits** or **decision rules**), by asking this questions we can divide our data in categories, data is divided on basis of information gain.
- true node -> internal nodes -> internal nodes or branches -> leave nodes
- if our leaf node has all examples in single class then its pure and if not then its impure
- how it works :
  - so first we check find entropy of root node
  - then from our dataset we choose the feature (these feature could be categorical or numerical)
  - for each feature, we find how much information gain it have.
  - we choose the feature with best gain
- **impurity :**
  - **Impurity** in the context of decision trees refers to **how mixed the classes are in a given node**. A node is **pure** if it contains only one class and **impure** if it contains multiple classes
  - impurity has three types Gini impurity (I_G), entropy (I_H), and the classification error (I_E).

  1. **Entropy :**
  - formulae :

  $$Entropy = -\sum_{i=1}^{C} p_i \log_2 p_i$$

  - Entropy = 0 → The node is **pure** (all samples belong to one class).
  - for pure class probabilities will be 0 and 1, for $\log_2 1$ is 0 and $0 * \log_2 0$ is also 0.
  - Entropy is high → The node has a **mixed** distribution of classes.
  - Entropy = 1 (for two classes) → The node is completely **impure** (equal distribution of both classes).
  - for $log_2 0.5$ = -1 and hence $-(0.5\log_2 0.5 + 0.5\log_2 0.5)$

  2. **Gini impurity :**
  - gini impurity It calculates the *probability of misclassifying a randomly chosen sample*
  - formulae for each leaf node

$$Gini = 1 - \sum_{i=1}^{C} p_i^2$$

- where c is total number of classes and p_i is the probability of
- for total gini impurity (for parent node) -> it is weighted average of gini impurities for the leaves

$$\sum_{c=1}^{n} \frac{N_c}{N_{total}} G_c$$

3. **classification error impurity :**
- formulae

$$Error = 1 - \max(p_i)$$

- we take the probability of class which have maximum probability and measure it misclassifying
-

- **information gain :**
  - the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurities of the child nodes, the larger the information gain.
  - formulae

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

here first term is impurity of parent node and second term is total impurity i.e weighted average of child impurities, D is dataset and N is number of examples

- information gain for binary tree classification

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

random forest

- random forest is combination of multiple decision trees, we use it because single tree has more chances to overfit while we combine them we reduce the risk of overfitting and it generalises better.

> Random sampling without replacement: 2, 1, 3, 4, 0
> Random sampling with replacement: 1, 3, 3, 4, 1

1. in random forest first we create bootstrap dataset by randomly selecting (with replacement) examples from training dataset. the size of bootstrap dataset is same as

original
2. then we randomly select subset of features from original set of features, num of features in this subset are less than original number of features a default in sickit-learn and other implementations is d = $\sqrt{m}$ where m is number of original features (The default `sqrt(m)` is for classification; for regression, scikit-learn uses `m/3` (or all features if `m <= 2`).)
3. then using these bootstraped dataset and subset of features we build decision tree
4. then we repeat this process and build multiple trees
5. we pass our data from multiple trees and each tree gives us one class (assuming our trees are classification trees) then we give ans/prediction a class that is given by most trees i.e final pred is pred given by most trees
6. while we build bootstrap dataset most of the elements are leftout as we do random sampling with replacement and hence all these samples that are not selected in bootstrap dataset are collected and this dataset is called out-of-bag dataset
7. to see how accurate our random forest is we pass our out-of-bag dataset from all trees. if model predicts wrong label the error is called out-of-bag error. and we get accuracy by taking proportion of errors with total samples

Example Walkthrough: OOB Error Calculation

Dataset

`[X₁, X₂, X₃]` (3 samples)

Tree Construction (Bootstrap Sampling)

1. **Tree 1**:
   - Bootstrap sample = `[X₁, X₂, X₂]` (sampled with replacement)
   - **OOB samples for Tree 1** = `{X₃}` (not selected in bootstrap)
2. **Tree 2**:
   - Bootstrap sample = `[X₁, X₃, X₃]`
   - **OOB samples for Tree 2** = `{X₂}`
3. **Tree 3**:
   - Bootstrap sample = `[X₂, X₃, X₃]`
   - **OOB samples for Tree 3** = `{X₁}`

   OOB Predictions

   For each sample, only trees where it was **OOB** vote:

- `X₁`:
  - OOB in **Tree 3** → Prediction = `Tree 3(X₁)`
- `X₂`:
  - OOB in **Tree 2** → Prediction = `Tree 2(X₂)`
- `X₃`:

    - OOB in **Tree 1** → Prediction = `Tree 1(X₃)`
    Error Calculation
    Assume:
- `Tree 3(X₁)` ≠ True label of `X₁` → **Error (1)**
- `Tree 2(X₂)` and `Tree 1(X₃)` are correct → **No error (0)**

OOB Error

$$\text{OOB Error} = \frac{\text{Number of misclassified OOB samples}}{\text{Total samples}} = \frac{1}{3} = 33.3\%$$

Key Notes
- Each sample is predicted **only by trees that did *not* see it during training** (OOB trees).
- OOB error estimates generalization **without a separate validation set**.

parametric non-parametric models

parametric models are models that have trainable parameters and we train and get values for this parameters from training dataset. in this method for predicting values for new dataset we don't need a training dataset, learned weights/parameters is all we need.
examples include : perceptron, logistic regression, linear SVM.

non-parametric models don't have fixed parameters for a model. we can have parameters that can change with each data like decision trees. or we might not have parameters at all like for knn.

k-nearest neighbours - lazy learning algorithm

in this algorithm we have training data with known labels. which is spread out in space. here we take new data point and then we check the labels of k (positive integer) nearest data points and then we assign the label that is most common in nearest data points

in this we don't have training phase as we don't have to train model its simple and hence also called lazy learning algorithm.

knn memorizes training dataset and is part of instance based learning.

1. Choose the number of k and a distance metric
2. Find the k-nearest neighbors of the data record that we want to classify
3. Assign the class label by majority vote

to find k-nearest neighbors we find the distance between our data point and all other datapoints using euclidian method and then chose nearest data point
formulae for euclidian distance

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sqrt{\sum_{k=1}^{n} |x_k^{(i)} - x_k^{(j)}|^2}$$

chapter 4 : Building Good Training Datasets – Data Preprocessing

`page no. 105 to 138`

colab notebook for all the code

imputing

scikit-learn resource

most of the time we get missing values in our data to deal with them we can remove them from dataset but it can cause loss of valuable information hence we use other methods like imputing

imputation is adding data on the spots where we missed data.
we can add mean, median or most occuring value in that feature column
mean imputation :

- here we simply replace the missing values with mean of the entire feature column to do this we can use `SimpleImputer` class in scikit-learn

estimaters and transformers

transformers are used to transform data like imputing missing values its an api of scikit-learn which consist classes that are used to transform data, it has two methods fit and transform fit is used to train model to transform data and transform is used to transform data

estimaters are use to make prediction on data they are trained on supervised datasets and then make prediction on testing data they have fit method and predict method they can also have transform method

categorical data

categorical data can be divide in two types ordinal and nominal.
`ordinal` data is when we can sort data in order for ex t-shirt size xl > l > m > s > xs
`nominal` data is when we can't sort it in order like t-shirt color

we have `df['col'].map(mapping dict)` to map values to entire column we use it to convert our ordinal data strings into numbers so our model can understand

converting categorical data to integers

- the data that is in strings in our dataset is need to convert into numbers so our model can understand it and to avoid any technical glitches
- we can do this by assigning numeric values to our strings by creating dict in python
- or we can use LabelEncoder class directly implemented in scikit-learn

one hot encoding method

- its a method to convert our categorical values into binary values
- in this the category for that example is set to be 1 and all other categories are set to be 0
- and in this way we create vector for each example

example:
Consider a categorical feature "Color" with three possible values:
["Red", "Green", "Blue"]

Color	One-Hot Encoding
Red	[1, 0, 0]
Green	[0, 1, 0]
Blue	[0, 0, 1]

- we have built in class in sickitlearn called `OneHotEncoder`

problem that emerges by using onehotencoding

using one hot encodings increases the number of columns which can be problem in methods where we have to inverse our matrix (matrices are computationally difficult to invert). in such cases we can remove the one feature

for example

sr.no.	price	size	color_green	color_red	color_blue
0	10.1	1	0	1	0
1	13.5	2	0	0	1
2	15.3	3	1	0	0

in above example if we remove one feature color blue we still don't lose valuable information because we know that when color green = 0 and color red = 0 , color blue will be 1

along with one hot encoding we have other methods to encode nominal categorical data

- binary encoding : its method where we convert our categorical data into integers and then we convert our integers into their binary representation and assign these binary representations to categories. its way more effective then one-hot encoding method.
- count or frequency encoding : its a method where we assign each numeric value to each category and this numeric value is number of times particular category is occured in our dataset

both of these methods have less cardinality (a large number of unique category labels) than one-hot encoding method

train test split of data

- we use `train_test_split` function from sklearn to split data
- we give it x values and y values and test_size
- x and y values should be arrays (numpy arrays)
- most commonly used splits are 60:40, 70:30, or 80:20
- for large datasets, 90:10 or 99:1 splits are also common and appropriate

feature scaling

- feature scaling is when we bring all features on same scale
- having scale difference can effect on our model in unintentional way like in gradient descent optimization algorithm
- decision trees and random forest algorithm are one of the very few algorithms that are not effected by scale
- there are two common approaches to bringing different features onto the same scale: normalization and standardization.
- **normalization**
    - in this we bring our feature in a range of [0,1] which is special case of min-max scaling
    - min-max scaling :
        - $$x_{\text{norm}}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$
        - where $x_{norm}^{(i)}$ is new value for example $x^{(i)}$
        - $x^{(i)}$ is a particular example, x_{min} is the smallest value in a feature column, and x_{max} is the largest value.
- **Standardization:**
    - Centers data at mean = 0 and scales to standard deviation = 1.
    - Makes feature columns resemble a standard normal distribution (but does not change the actual distribution shape).

- Preferred for optimization algorithms (e.g., gradient descent) and linear models (e.g., logistic regression, SVM).
- Helps with weight learning because many models initialize weights near 0—standardized features align better with this initialization.
- Less sensitive to outliers compared to min-max scaling.
- Key Notes:
    - Standardization does not convert non-normal data into a normal distribution.
    - It preserves information about outliers better than min-max scaling.
    - Many machine learning algorithms perform better with standardized features.
- formulae for standardization

$$x_{\text{std}}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Here, μ_x is the sample mean of a particular feature column, and σ_x is the corresponding standard

example of both normalizaition and standardization

Input	Standardized	Min-max normalized
0.0	-1.46385	0.0
1.0	-0.87831	0.2
2.0	-0.29277	0.4
3.0	0.29277	0.6
4.0	0.87831	0.8
5.0	1.46385	1.0

- we use different function available in sklearn to standardize data like `StandardScaler` is commonly used
- we also have `RobustScaler` which is especially helpful and recommended if we are working with small datasets that contain many outliers.
- Operating on each feature column independently, RobustScaler removes the median value and scales the dataset according to the 1st and 3rd quartile of the dataset (that is, the 25th and 75th quantile, respectively) such that more extreme values and outliers become less pronounced.

solving overfitting

- when model has more test loss and training loss i.e model performs better on training data but performs poorly on testing data it happens due to overfitting our model is memmorizing data instead of generalizing. on it we call it our model has high variance.

- the reason for overfitting can be that our model is too complex for given data common methods to solve overfitting model are as follows :
  - Collect more training data
  - introduce a penalty for complexity via regularization
  - Choose a simpler model with fewer parameters
  - Reduce the dimensionality of the data

regularization

- when our model overfit we add penalty term in our loss so we can punish model for extreme values (outliers) and in this way we can solve overfitting problem
- there two ways L1 and L2

L2 regulaization (Ridge Regularization)

$$L2 : \|w\|_2^2 = \sum_{j=1}^{m} w_j^2$$

- **for linear regressiono** our weights act as slope of line we try to fit for data. our size of weight determines how much it wil impact on result, the more the size of weight (slope) more will be the size of result i.e result will be more sensitive for weight size, and L2 tries to keep the size of slope relatively less so that our weights don't effect much on results
- larger the size of λ gets, smaller the size of slope gets i.e they are inversely proportional

> A larger λ means a stronger penalty on the magnitude of the weights. To minimize the overall cost function (which includes the loss from the data and the regularization penalty), the model is forced to choose smaller weight values.

L1 regularization (Lasso Regularization)

$$L1 : \|w\|_1 = \sum_{j=1}^{m} |w_j|$$

- in lasso regularization just like ridge regularization we try to minimize our parameters so that our line (decision boundry) is not dependent on weights, but difference is here we can shrink our parameters to zero (once which do not contribute much on deciding results) which we can't do in ridge regression
- L1 is used when we have too many features and most of them are irrelevant and L2 is used when most of the feature are relevant and impact on output
- L1 converts most feature vectors to zero this is also called sparsity and Sparsity can be useful in practice if we have a high-dimensional dataset with many features that are irrelevant, especially in cases where we have more irrelevant dimensions than training examples

feature selection

- in this we select a subset of the original features to reduce the number of feature and hence to reduce dimensionality of data
- Sequential feature selection algorithms are a family of greedy search algorithms that are used to reduce an initial d-dimensional feature space to a k-dimensional feature subspace where k < d
- we use feature selection so we can reduce the amount of computation cost and to reduce the overfitting in models that don't support regulization

sequential backward selection

- sbs removes features sequentially until we have only useful features remaining to decide which feature to remove we decide on criterion J which we want to minimize
- so we remove feature one by one from our total features and then check how much performance changes if our performance increases (or only slightly loss happens) after removing that feature we remove that feature and if our performance decreases (major loss change) then we keep it.
- so we keep the feature that has smallest loss than others
- stepwise algorithm :

1. Initialize the algorithm with $k = d$, where d is the dimensionality of the full feature space, X_d.
2. determin the feature (x^-) which maximizes the criterion: $x^- = argmax J(X_k - x)$ where $x \in X_k$
3. Remove the feature, (x^-), from the feature set: $(X_{k-1} = X_k - x^-); (k = k - 1)$.
4. Terminate if k equals the number of desired features; otherwise, go to step 2.

- we can use `SelectFromModel` object in sklearn to do feature selection using RandomForest (randomforest can be used to select features as it know which features are important) as feature selector

chapter 5 : Compressing Data via Dimensionality Reduction

`page no. 139 to 170`
colab notebook

overview

- linear vs non-linear data compression
- linear :
  - PCA : principal component analysis (class labels are not considered)
  - LDA : linear discriminant analysis (supervised, class labels are considered)

- non-linear :
  - t-SNE : t-distributed stochastic neighbor embedding

PCA : principal component analysis

- its used in unsupervised dimensionality reduction
- in feature extraction we transform our data onto new feature space on other hand in feature selection we only remove non-important features and other feature remain same
- its compressing data without losing relevant and important information
- reduces the curse of dimensionality

main idea

- we have input vector $x = [x_1, x_2, \ldots, x_d]$ of dimension d
- we want to convert it into dimension k where $k < d$
- so we use transformation matrix W
- such that $xW = z$
- where $z = [z_1, z_2, \ldots, z_k]$, is vector of k dimensions

  notes
- first we find principal component i.e axis around which data is has most variance
- then we find other pc2 which is perpendicular to pc1 and is axis where data is most variance after pc1
- standardization is important in pca as feature scale impact heavily on results

detailed steps

1. standardize the data (its important that our features are on same scale)
2. find covariance between features :
   - formulae :

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^{n} \left(x_j^{(i)} - \mu_j \right) \left(x_k^{(i)} - \mu_k \right)$$

   - μ_j is mean of jth feature and same for μ_k. so basically σ_{jk} gives us one numeric value which gives us the covariance between ith and jth value.
   - also mean is zero for standardized data
3. create a matrix of these covariances

$$C = \begin{bmatrix} \sigma_{12} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{32} \end{bmatrix}$$

4. then we find eigen vector and eigen values for this covariance matrix :
   - eigen vector satisfy's following formulae : $Cv = \lambda v$ where λ is scaler eigen value

- first we find eigen values $det(C - \lambda I) = 0$, where λ is eigen values we need to find and I is identity matrix of same size of C
- then for each eigen value we find corresponding eigen vector by using equation $(C - \lambda I)v = 0$, where $\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{and} \quad \mathbf{0} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$
- eigen value gives the magnitude of spread (higher the eigen value, more the spread or variance) or variance and eigen vector associated with gives us the values of direction
- so after this process we get eigen values and eigen vectors associated with them

5. we arrange our eigen values from greatest to lowest and then chose eigen vectors associated with first k (number of dimensions we want to convert our data onto) eigen values
   - The choice of k depends on how much of the total variance you want to retain. A common method is to look at the cumulative explained variance:

$$\text{Cumulative Explained Variance}_k = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{j=1}^{p} \lambda_j}$$

6. Create a projection matrix W (of size $d \times k$) by stacking these top k eigenvectors as columns:

$$W = [v_1, v_2, \dots v_k]$$

7. Transform the Original Data:

$$Y = BW$$

   here B is the original data matrix and Y is new data matrix (with reduced dimensinality)

intuition for covariance

- Now, for each data point i, we multiply the deviation of the j-th feature (b_{ij}) by the deviation of the k-th feature (b_{ik}). Let's consider the possible scenarios:
  - **Both features are above their means:** If b_{ij}>0 and b_{ik}>0, then their product $b_{ij}b_{ik}$ is positive. This suggests that when the j-th feature is high, the k-th feature also tends to be high.
  - **Both features are below their means:** If $b_{ij} < 0$ and $b_{ik} < 0$, then their product $b_{ij}b_{ik}$is also positive (negative times negative is positive). This suggests that when the j-th feature is low, the k-th feature also tends to be low.
  - **One feature is above its mean, the other is below:** If $b_{ij} > 0$ and $b_{ik} < 0$ (or vice versa), then their product $b_{ij}b_{ik}$ is negative. This suggests that when one feature is high, the other tends to be low (an inverse relationship).
  - **One or both features are at their means:** If $b_{ij} = 0$ or $b_{ik} = 0$, then their product is zero, indicating no deviation from the mean for that point in that feature.

LDA : linear discriminant analysis

linear discriminant analysis aims to find the directions (linear discriminants) that **maximize the separation between the means of different classes**, while simultaneously **minimizing the variance within each individual class**.

difference between pca and lda :

- so while pca tries to find overall most spread between entire dataset lda tries to find most spread between different classes only and it tries to minnimize the spread inside the class
- pca is unsupervised and lda is supervised

detailed steps

1. standardize the data (important because we want our data to be in same scale)
2. we first find mean matrix of each of class
    - for class i we calculate the mean of all examples that are in this class
    - each example is vector of features represented like this $x_{ij} = [x_{ij1}, x_{ij2}, \ldots, x_{ijn}]$ i.e i^{th} example with j features
    - formulae

$$\mu_i = \frac{1}{N_i} \sum_{j=1}^{N_i} x_{ij}$$

    - where u_i is the mean vector for that class N_i is the number of examples in that i class
3. next step is we calculate within class scatter matrix :
    - here we calculate spread of data within a class
    - we use following formulae to calculate scatter matrix for each class i

$$S_i = \sum_{j=1}^{N_i} (\mathbf{x}_{ij} - \mu_i)(\mathbf{x}_{ij} - \mu_i)^T$$

- here $(\mathbf{x}_{ij} - \mu_i)$ gives us how for are the data point from its mean within that class
- and we do $(\mathbf{x}_{ij} - \mu_i)(\mathbf{x}_{ij} - \mu_i)^T$ this multiplication because if we multiply column vector with its transpose it gives us square matrix of $n \times n$
- in this $n \times n$ scatter matrix diagonal elements represents spread of data of each feature along with its mean an off diagonal elements represents spread of data compared with each feature
- we further sum these scatter matrices of classes to get one within class scatter matrix S_W using formula

$$S_W = \sum_{i=1}^{C} S_i = \sum_{i=1}^{C} \sum_{j=1}^{N_i} (\mathbf{x}_{ij} - \mu_i)(\mathbf{x}_{ij} - \mu_i)^T$$

```
- having different number of examples for each class can impact our
scatter matrix and hence in such cases we normalise above formula
```

$$S_i = \frac{1}{n_i} S_i = \frac{1}{n_i} \sum_{j=1}^{N_i} (\mathbf{x}_{ij} - \mu_i)(\mathbf{x}_{ij} - \mu_i)^T$$

4. next we find between-class scatter matrix:
   1. first we find overall mean of entire dataset so we can then check how far are other means of different classes from main mean of entire dataset. to do this we use given formula

   $$\mu = \frac{1}{N} \sum_{i=1}^{C} N_i \mu_i$$

   where μ is mean of entire dataset and μ_i is the mean of each class i
   - so then between-class scatter matrix is then calculated using formula

   $$S_B = \sum_{i=1}^{C} N_i (\mu_i - \mu)(\mu_i - \mu)^T$$

   - now working of this matrix is also same as within-class scatter matrix but instead of datapoints within same class here we get relationships (spread/variance) between different calsses
5. now that we have both S_W and S_B we find eigen value and eigen vector of $S_W^{-1} S_B$
6. now we select the k eigen vectors corresponding to k largest eigen values
   - The eigenvectors w represent the directions (linear combinations of the original features) that best separate the classes. The eigenvalues λ indicate the significance of each eigenvector – the larger the eigenvalue, the more discriminatory information is captured by its corresponding eigenvector.
   - These eigenvectors form the columns of a transformation matrix W of size $n \times k$:

   $$W = [w_1, w_2, \ldots, w_k]$$

7. finally we project our data onto the new subspace using formula

   $$y = W^T x$$

   - The resulting y is a vector of size $k \times 1$

chapter 6 : Learning Best Practices for Model Evaluation and Hyperparameter Tuning

`page no. 171 to 204`

colab notebook for all code

overview

- building a pipeline (transformation techniques and classifiers combined)
  - combining data preprocessing with model training and prediction
  - scaling, dimensionality reduction, model fitting, and model transform
- assessing model performance using holdout cross-validation and k-fold cross-validation
  - first dividing dataset into training and testing part
  - and then dividing that training dataset into training and validation dataset
  - reason -> testing ur model on test dataset for multiple times can cause model to learn testing data and can cause overfitting issues
- learning and validation curves
  - using these curves to find what's the problem with model like overfitting vs under-fitting
- hyper-parameter optimization techniques
  - grid search
  - randomized search
  - search with successive halving
- nested cross validation
- performance evaluation metrics
  - confusion matrix
  - precision and recall of classification models
  - receiver operating characteristic graph
  - multiclass scoring metrics
- dealing with class imbalance
  - resampling

pipelines

- its a wrapper tool used to combine transformers and estimators in single pipeline.
- it uses pipeline object by scikit-learn and we just have to give our transformers and estimators to that object as parameters.
- their is no limit on how many transformers we can have but in the end we want to have estimator to do the prediction
- we can use `.fit` and `.predict` methods on our pipeline object `.fit` will use `.fir_transform` on transformers and `.fit` on estimators
- for `.predict` it uses `.transform` on transformers and `.predict` on estimators
- in this pipeline data is basically passed from one function to another (all these functions are the ones that we give as parameters to pipeline object)

cross-validation techniques

the holdout method

- in this method first we split our data in training and testing part and letter on we split the training part into training and validation part.
- we do this because before testing our data we have to check which model or hyperparameters work best for our problem and the process of choosing best model is called model selection. *if we keep checking our model on testing data, then testing data will become part of our training i.e our model will learn testing data too and it will increase the chances of model overfitting.*
- we use test dataset to find model's performance on unseen data.
- the drawback of this method is our validation data is fixed and our results will highly depend on what kind of validation data we have.

k-fold cross-validation

- in this method we divide our training dataset in to k-folds i.e k-parts.
- we use k-1 folds to train our model this k-1 folds of data is also called training fold. and we use that remaining 1 fold as test fold to evaluate our model.
- we repeat this process for k times i.e until our each fold acts as test fold.
- the we take average estimation of results we got for each iteration.
- this method is used so our result won't be heavily dependent on that one validation dataset.
- after getting the best results for particular hyperparameters we again train our data on entire training dataset and test it on test dataset to find how good it generalizes on unseen data.
- good value of k is 10 (For instance, experiments by Ron Kohavi on various real-world datasets suggest that 10-fold cross-validation offers the best tradeoff between bias and variance).
- if we are working with smaller dataset we use large value of k so that we can get more data for training but it also comes with risk of more runtime and high variance as training data will be more and more similar and when we are working with large dataset we use small value of k like 5 to keep the computational cost low.
- there is also special varient of this technique called Leave-one-out cross-validation where we set k = number of examples in training dataset, so only 1 example will be used as test fold. its use when working with extremely small dataset.
- slightly improved version of this method is stratified cross-validation in this the class label proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset.

learning and validation curves

learning curves and validation curves helps us identify common problems in ml like under fitting and over fitting

- under fitting
  - here our model performs worse on both training and validation phase.
    - in this case we can increase our parameters by collecting additional features or decrease the strength of regularization parameter.
- overfitting
  - this happens when our model is too complex for simple training data.
  - in such cases it helps to increase the training data, reduce the complexity of model or increase the regularization strength. for unregularized models we can decrease the number of features.
- learning curves
  - in learning curves we plot our training and validation accuracy against number of training examples in our dataset.
- validation curves
  - here we plot our accuracies against values of model parameters for example inverse regularization parameter C, in logistic regression.
- Similar to the learning_curve function, the validation_curve function uses stratified k-fold cross-validation by default to estimate the performance of the classifier.

hyperparameters optimization techniques

there are two types of parameter in machine learning one that are trained during training process from the training data, for example, weights in logistic regression, and the parameters that are trained separately also called hyperparameters, for example, regularization parameter in logistic regression.

some of the techniques to optimize hyperparameters are given below :

grid search

- in this algorithm we train our model for each combination of specified hyperparameters and then choose the combination of parameter that provided us the best score.
- to evaluate the model we use cross-validation technique.
- we create the list containing dictionaries and each dictionary contains different combination of hyper-parameters.
- it’s a brute-force exhaustive search paradigm.

randomized search

- grid search is guaranteed to find the optimal hyperparameters out of the given list combinations of hyoperparameters, but we can miss the most optimal combinations sometimes and increasing size of grid (list of combination of parameters) is computaionally not feasible.
- to solve the above problem we use another technique called **randomized search**.

- in this search we specify the number of parameters we want to sample from given list(in GS this list is small and discrete while in RS we have list of large numbers like continuous list of numbers between (0.0001, 1000.0).)
- from this list we randomly sample the number of parameters that we have specified.
- in this method, instead of specifying a discrete list of values for each hyperparameter (as in Grid Search), you define a *distribution* or a *range* from which values for each hyperparameter can be sampled.

successive halving approach for grid search and randomized search

- in this technique we start with the list of hyperparameters (it could be randomized for RS or user defined values for GS)
- first we train model with limited sources, for example, a small subset of training data instead of entire dataset
- then discard the bottom 50 percent on the basis of their score
- we again repeat step no. 2 this time with more resources
- this process is repeated until only one hyperparameter configuratioon remains.

nested cross-validation

- in nested cross-validation we have two loops, first loop performs the k-fold cross-validation on entire dataset.
- for each iteration of first loop, we use k-fold cross validation on the training dataset of that iteration, and then we do final evaluation for that iteration on test dataset of that particular iteration
- given is a diagram for this :

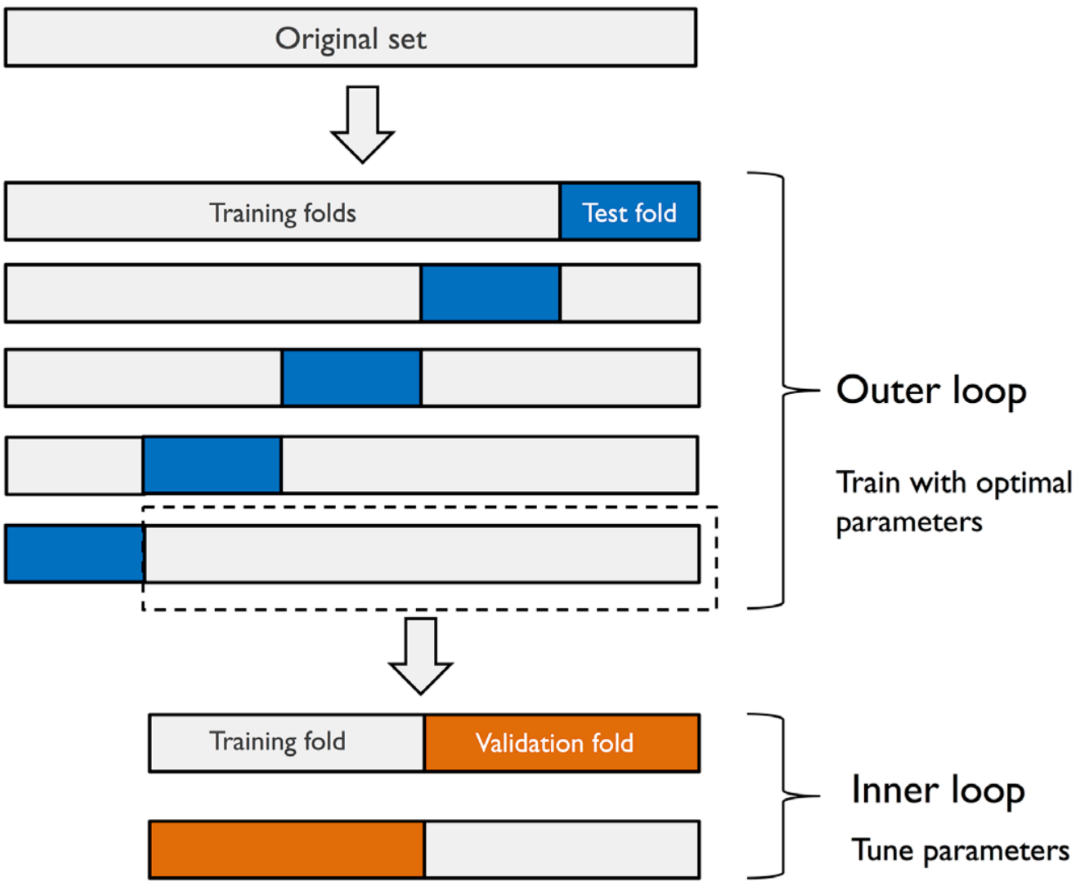


Figure 6.8: The concept of nested cross-validation

performance evolution metrics

confusion matrix, recall, precision, f1 score and MCC

confusion matrix is a matrix that contains true positives, true negatives, false positives, and false negatives.

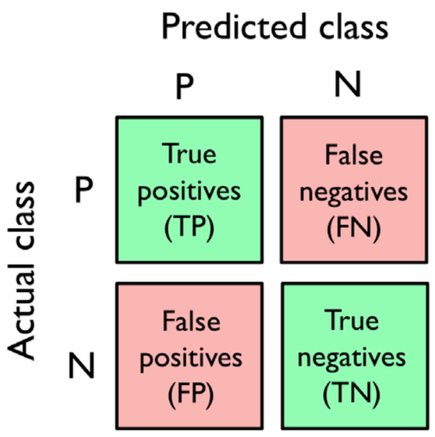


Figure 6.9: The confusion matrix

The error can be understood as the sum of all false predictions divided by the number of total predictions, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions.

formulae for error and accuracy

$$\mathrm{ERR} = \frac{\mathrm{FP} + \mathrm{FN}}{\mathrm{FP} + \mathrm{FN} + \mathrm{TP} + \mathrm{TN}}$$

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{FP} + \mathrm{FN} + \mathrm{TP} + \mathrm{TN}} = 1 - \mathrm{ERR}$$

The **true positive rate (TPR)** and **false positive rate (FPR)** are performance metrics that are especially useful for imbalanced class problems:

$$\mathrm{FPR} = \frac{\mathrm{FP}}{N} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

$$\mathrm{TPR} = \frac{\mathrm{TP}}{P} = \frac{\mathrm{TP}}{\mathrm{FN} + \mathrm{TP}}$$

recall quantifies how many of the relevant records (the positives) are captured as such (the true positives).

$$\mathrm{REC} = \mathrm{TPR} = \frac{\mathrm{TP}}{P} = \frac{\mathrm{TP}}{\mathrm{FN} + \mathrm{TP}} \quad \text{where} \quad P = \mathrm{FN} + \mathrm{TP}$$

Precision quantifies how many of the records predicted as relevant (the sum of true and false positives) are actually relevant (true positives).

$$\mathrm{PRE} = \frac{\mathrm{TP}}{\mathrm{FP} + \mathrm{TP}}$$

To balance the up- and downsides of optimizing PRE and REC, the harmonic mean of PRE and REC is used, the so-called **F1 score**:

$$F1 = 2 \cdot \frac{\mathrm{PRE} \times \mathrm{REC}}{\mathrm{PRE} + \mathrm{REC}}$$

a measure that summarizes a confusion matrix is the **MCC**, which is especially popular in biological research contexts. The MCC (Matthews correlation coefficient) is calculated as follows:

$$\mathrm{MCC} = \frac{\mathrm{TP} \times \mathrm{TN} - \mathrm{FP} \times \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})}}$$

in the gridsearch or randomized search instead of accuracy as score we can provide any of the above criterias as score

how precision and recall is needed according to problem in real world :
For instance, our priority might be to identify the majority of patients with malignant cancer to recommend an additional screening, so recall should be our metric of choice. In spam

filtering, where we don’t want to label emails as spam if the system is not very certain, precision might be a more appropriate metric.

receiver operating characteristic (ROC graphs)

- so roc we use TPR and FPR to plot how good our model predictions are
- we plot this graph by shifting the decision boundary each time and calculating TPR and FPR.
- The diagonal of a ROC graph can be interpreted as random guessing, and classification models that fall below the diagonal are considered as worse than random guessing.
- A perfect classifier would fall into the top-left corner of the graph with a TPR of 1 and an FPR of 0.
- Based on the ROC curve, we can then compute the so-called ROC area under the curve (ROC AUC) to characterize the performance of a classification model.
- more the AUC, better the model is performing.

performance metrics for multi-class classification

- for multiclasses we use the same scoring techniqnues like precision, recall etc. but with one vs all approach i.e finding TP, FP, TN, FN for each class individually.
- you treat one class as positive class and all other classes as negative classes and find TP, FP, TN, FN for that one class and repeat that same method for other classes. this particular method is called **macro average**, here in this method we find precision (or any other score) individually for each class and then sum those precisions and divide them by total number of classes. we use this method when our classes are balanced because class imbalance can effect in this method.
- **micro method** : here we first find TP, FP, TN, FN of each class and then sum them up to find global TP, global FP, global TN, global FN, and then we find precision of them. its best for when dataset is imbalanced or when we care about over correctness.
- **weighted average** : here we first find precision, recall, and F1-score independently for each class. and then we take average of these scores per class, weighted by the number of true instances for each class.

class imbalance and techniques to deal with it

- class imbalance is when we have example with different number of classes, for example, say we have two classes, class A and class B, class will be imbalanced when class A will have 90 examples and class B will have only 10 examples.
- this can create a problem as even if we guess only one class value in our model, it will give us 90% accuracy. hence we don't use accuracy as scoring metric and rather we can use different metrics like precision, recall, F1 score or MCC score.
- one way to deal with this is to assign larger penalty to wrong predictions on the minority class. we can do this in scikit learn by changing the `class_weight` parameter, by

default this parameter is set to `class_weight='balanced'`.
- other strategies include upsampling the minority class, downsampling the majority class, and the generation of synthetic training examples.

chapter 7 : Combining Different Models for Ensemble Learning

`page no. 205 to 246`

colab notebook for all code

overview

- what is ensemble?
  - its when u combine multiple models for single problem, for example, combining logistic regression and knn to solve some binary classification problem.
  - majority voting principle :
    - it simply means predicting class label that is predicted by more than 50 percent of classifiers.
  - building majority vote classifier i.e ensemble multiple models with majority vote principle.
- evaluating and tuning the ensemble classifier :
  - using roc curve to evaluate our model.
  - using methods like gridsearch to tune parameters of our model for better performance.
- bagging :
  - in this method we sample a data from training dataset, for each individual classifier separately with replacement. i.e data is sampled with replacement from training data set to train each classifier.
- boosting
  - in this method we take some weak classifiers (they have only slight advantage over random guessing), we train them to learn from their misclassified training example to improve performance.
  - adaptive boosting (AdaBoost)
    - concept
    - implementation using scikit-learn.
  - gradient boosting
    - concept
    - comparison with adaboost
    - implementation : XGBoost

ensemble method

- in this method we combine multiple classifiers to create a meta classifier which would outperform these individual classifiers
- in ensemble method we combine multiple classifiers and each classifier gives its prediction, to choose one final outcome we need a method to choose one prediction and the method we gonna use is called **majority vote prediction method**.
- majority vote strictly speaking is binary classification method but it can be easily generalised for multiclass classification also know as **plurality voting**.
- in majority vote prediction we choose the prediction that is predicted by most of the classifiers, for example, lets say we have 5 classifiers, and 3 classifiers gives prediction as class 1, and 2 classifiers gives prediction as class 0. then our final prediction will be class 1.
- ensembled classifier generally have lower error rate than individual classifiers. we can prove this by calculating pmf of binomial distribution.
- The term 'majority voting' is used interchangeably with 'plurality voting' in the literature, so we will use the term 'majority voting' even when we are actually using plurality voting.
- **why ensemble method works** :
  - If you have N independent models each with error rate ε, the majority vote error is much lower than ε. It's like asking multiple experts instead of just one - even if each expert is sometimes wrong, the group decision is usually better.

majority voting

- so in simple majority voting we only look for class that is predicted by most of the classifiers and make it our final answer
- there are other methods of majority voting :
- **weighted majority voting** :
  - in this method we assign weight to each classifier.
  - we check for each class, if any classifier predicts that class then we add weight for that class, and we do this for every classifier and every class.
  - claude's explanation : We assign a weight to each classifier based on how much we trust it. Then, for every possible class, we calculate a total score by adding up the weights of all classifiers that predicted that specific class. The class with the highest total weighted score becomes our final ensemble prediction.
  - for example : we have two classes [0,1] and three classifiers and weights assigned to each classifier $C_1 : 0.2, C_2 : 0.2, C_3 : 0.6$, let's say first two classifiers C_1 and C_2 predicts the class 0 and C_3 predicts the class 1, then we calculate for class 0 = 0.2+0.2 = 0.4 (adding weights for class) and for class 1 = 0.6 , then we check which class has more weight, class 0 = 0.4 < class 1 = 0.6, hence final prediction will be class 1. on the other hand if we had chosen normal majority voting method it would have predicted class 0 as answer as it assumes each classifier holds equal weight.
  - weight means how much we trust the classifier or how accurate classifiers is, more weight indicated that classifier is good at predicting.

- **majority voting using probabilities instead of class label**
  - in this method we use class probabilities instead of predicted labels.
  - formula :

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

  - here p_j is probability for ith class and w_j is the weight for that classifier.
  - for example :
  To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0, 1\}$ and an ensemble of three classifiers, $C_j (j \in \{1, 2, 3\})$. Let's assume that the classifiers C_j return the following class membership probabilities for a particular example, x:

$$C_1(x) \rightarrow [0.9, 0.1], \quad C_2(x) \rightarrow [0.8, 0.2], \quad C_3(x) \rightarrow [0.4, 0.6]$$

Using the same weights as previously (0.2, 0.2, and 0.6), we can then calculate the individual class probabilities as follows:

$$p(i_0|x) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$
$$p(i_1|x) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$
$$\hat{y} = \arg\max[p(i_0|x), p(i_1|x)] = 0$$

prediction is class 0.

bagging

- its used in majority voting classifier, here we draw bootstrap samples (random samples with replacement) for each classifier.
- bootstrap samples : let's say we have training dataset with 100 samples then we will randomly choose 100 samples from this training dataset, but after choosing each sample we will keep it in dataset while going for next sample, which causes our new dataset to have duplicates and this is what we call bootstrap sampling. irl our sampled dataset consist 63% unique samples and other duplicated.
- in majority vote classifier, for each classifier we create new dataset by using bootstrap sampling and use that dataset to train our classifier and predictions are combined using majority voting, this process is called bagging.
- bagging is better because it reduces the chance of our classifiers overfitting to the data.

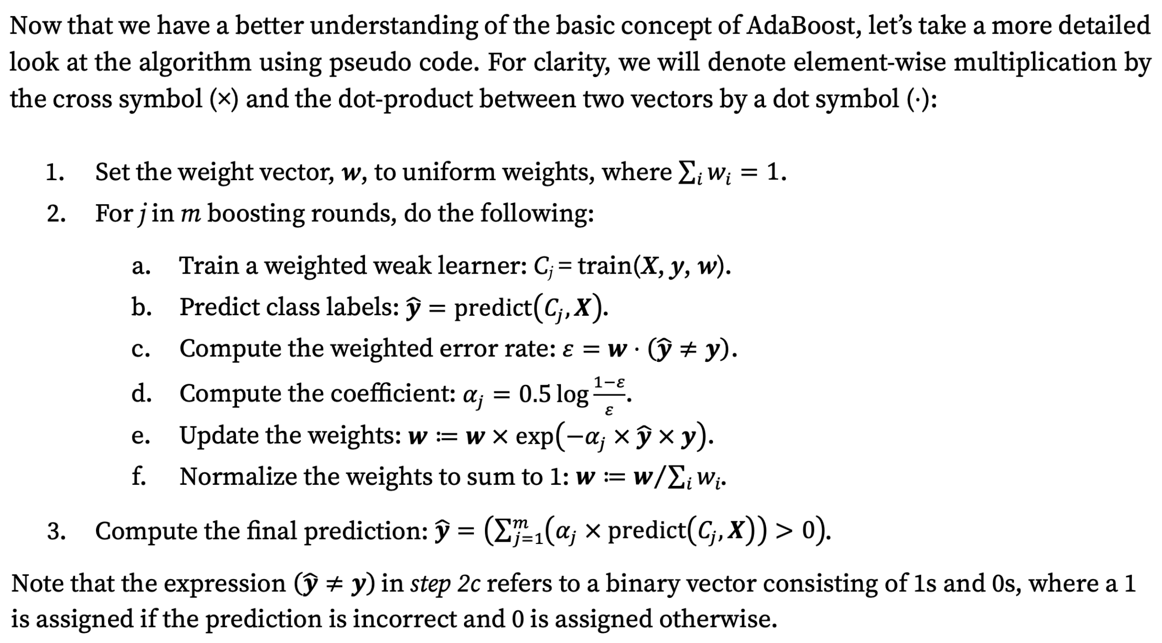
boosting

- boosting is the method where we use weak learners to learn from misclassified examples.
- weak learner : these are models that are only slightly better than random guessing.
- how it works :

- Train a weak learner
  - Identify which examples it got wrong
  - Train the next model to focus more on those mistakes
  - Repeat this process
  - Combine all models with weighted voting
- let's see example :
  - Draw a random subset (sample) of training examples, d_1, without replacement from the training dataset, D, to train a weak learner, c_1.
  - Draw a second random training subset, d_2, without replacement from the training dataset and add 50 percent of the examples that were previously misclassified to train a weak learner, c_2.
  - Find the training examples, d_3, in the training dataset, D, which c_1 and c_2 disagree upon, to train a third weak learner, c_3.
  - Combine the weak learners $c_1, c_2 \text{ and } c_3$ via majority voting.
- given are the applications of boosting technique : Adaboost, gradient boosting and XGboost.

Adaboost

- first we create a weight vector and assign equal weights to each our example.
- then we train our weak learner on entire dataset.
- we make predictions.
- for wrong predictions we increase the weights associated with that example, and for right prediction we decrease the weights.
- then we repeat this process with new weights and new weak learner for all our classifiers.
- and then we use majority voting to make the final prediction.
- below is the detailed process of adaboost :



Now that we have a better understanding of the basic concept of AdaBoost, let’s take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol (×) and the dot-product between two vectors by a dot symbol (·):

```
1. Set the weight vector, $\boldsymbol{w}$, to uniform weights, where $\sum_i w_i = 1$.
2. For $j$ in $m$ boosting rounds, do the following:
    a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
    b. Predict class labels: $\hat{\boldsymbol{y}} = \text{predict}(C_j, \boldsymbol{X})$.
    c. Compute the weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.
    d. Compute the coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
    e. Update the weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
    f. Normalize the weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
3. Compute the final prediction: $\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} \left(\alpha_j \times \text{predict}(C_j, \boldsymbol{X})\right) > 0\right)$.
```

Note that the expression $(\hat{\boldsymbol{y}} \neq \boldsymbol{y})$ in *step 2c* refers to a binary vector consisting of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

- so the coefficient α_j is the weight associated with each classifier for majority voting.
- so the main function of Adaboost is to find weights for each classifier so we can perform weighted majority voting technique on it.
- *in adaboost our weights effect how decision tree calculates loss or entropy hence when our example have more weight associated with each it will increase our entropy and our decision tree algorithm will focus more on predicting that example correctly. and this is why when we increase the weights of our wrong example the next weak learner will focus on getting those right*

example from claude on adaboost

Gradient boosting

- **comparing it with adaboost**
    - in adaboost we train decision tree stump (single level decision tree) and use weights associated with each to find error rate i.e weight for classifier and we change this weight for examples that are wrong.
    - in gradient boosting we use decision trees but with more depth typically max depth of 3 to 6.
    - instead of having an individual weighting term for each tree, like in AdaBoost, gradient boosting uses a global learning rate that is the same for each tree.
- **outline**
    - In Gradient Boosting, the gradient is not used to update model parameters. Instead, the gradient for each data point becomes the **target that the next decision tree is trained to predict.**
    - we don't have weights and we add predictions from each tree instead of using majority votiing
    - the final prediction might be sum of predictions from all the trees and it will be calculated for each example
    - step 0 :
        - before starting to do predictions on data using any decision trees we need a base model which will make errors and we have to correct those errors in upcoming trees.
        - so this model gives a prediction $F_o(X)$ which is constant for all the example in our training dataset.
        - we find this prediction using log(odds) formula for true class (y = 1).
        - formula :

$$\hat{y} = \log(\text{odds}) = \log\left(\frac{p}{1-p}\right)$$

        - here p is the probability of class y.
        - lets say we have 10 example in dataset, 6 of them are from class 1 and 4 from

class 0, then p = 6/10 = 0.6
    - and odds = 0.6 / 0.4 = 1.5
    - and ŷ = log(1.5) ≈ 0.405
    - now this $\hat{y} = 0.405$ will be the constant prediction for all our examples.
- step 1 :
    - so here we first convert our log(odds) or $\hat{y}$ into probabilities using sigmoid function $p_i = \frac{e^{F_0(x_i)}}{1+e^{F_0(x_i)}}$
    - calculate the gradient for each data point using the formula = $\frac{\partial L_i}{\partial \hat{y}_i} = p_i - y_i$.
    - Remember, gradient descent tells us to move in the *opposite* direction of the gradient. So, the "error" or "residual" we want to correct for each data point is the **negative gradient** which is also called **Pseudo-residual**, $r_{i1} = y_i - p_i$.
        - intuition:
        - If the true label was 1 and we predicted a low probability (e.g., 0.2), the residual is $1 - 0.2 = 0.8$. The next tree needs to learn to add a positive value.
        - If the true label was 0 and we predicted a high probability (e.g., 0.7), the residual is $0 - 0.7 = -0.7$. The next tree needs to learn to add a negative value.
    - this pseudo-residual value is the target for our next decision tree which it has to predict.
- step 2 :
    - this is where we train our decision tree (h_1).
    - here our inputs are same from our training dataset i.e inputs = X
    - but target variables is NOT the original y_i (0s and 1s). The target is the set of **pseudo-residuals**, r_{i1}, that we just calculated.
    - and we train our tree h_1 on this new data, this tree gives us new predictions $h_1(X)$ i.e new pseudo-residuals.
- step 3 :
    - in this step we add our next tree h_2.
    - just like previous tree inputs for it are same as in our training dataset X.
    - but for outputs we don't replace our previous pseudo-residuals with new pseudo-residuals but instead we add these new preds into our previous pseudo-residuals.
    - we use given formula for it :

$$F_1(x) = F_0(x) + \nu \cdot h_1(x)$$

- Step 4: Repeat for the Next Tree
    - Now, we do it all over again.
    - Use the updated model $F_1(x)$ to calculate new probabilities (p_i) and new pseudo-residuals $(r_{i2} = y_i - p_i)$. These residuals will be smaller because our model $F_1(x)$ is better than $F_0(x)$.

- Train a *second* tree (h_2) to predict these *new* residuals (r_i2).
- Update the overall model again: $F_2(x) = F_1(x) + \nu \cdot h_2(x)$.
- This process is repeated for hundreds or thousands of trees, with each new tree focusing on correcting the remaining errors of the entire ensemble that came before it.

chapter 8 : Applying Machine Learning to Sentiment Analysis

`page no. 247 to 268`

colab notebook for all code

overview

- sentimental analysis is also know as opinion mining.
- downloading and accessing data, converting it into csv and then accessing it through df's
- learning about bag of word model, term frequencies, tf-idf technique
- cleaning text data : removing punctuations and unwanted symbols
- coverting text into tokens : using porter stemmer algorithm to tokenize text, also removing stop words
- training logistic regression model for classification tasks, using gridsearch to find best parameters for our model
- working with bigger data : online algorithms and out of core learning, hashing vectorizer
- text and LDA

random intuition

when we do sentimental analysis on dataset that consists movie reviews as input feature and sentiment as 0 or 1 number i.e representing positive and negative we use tf-idf vectorization method convert inputs into vectors these vectors have frequencies of words for each document, we use these vectors as input and use logistic regression to predict o or 1. so my intuition was that model will see and remember like specific word index i.e word from vocab and if appears like multiple examples that are negative i.e class 0 then model will predict class 0 if our example have that word in review i.e our vector have large number on that word's index and this way our model learns multiple patters and make predictions.

data preparation

- downloaded dataset in colab
- using nested for loop to access the files and read data from them
- save that data to panda's dataframe object
- then reindexing and converting that dataframe to csv file

- the process of data preparation is highly dependent on what king of data ur downloading and in what format so there is no fixed process to prepare data we just know the end format in which we want our data and thats it.

bag-of-words model

- allows us to represent text as numerical feature vector.

principle behind bag-of-words :

- We create a vocabulary of unique tokens—for example, words—from the entire set of documents.
- We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.
- as we are creating unique tokens vocabulary, we are creating it from all the set of documents i.e all our training dataset, but when we create a feature vector we only consider the frequency of that word in that particular document, so we know that each document will have only subset of words from our entire unique token vocab, so other words will be zero for that document, which is why we call them sparse.

raw term frequencies :

so first we create a token vocabulary : this vocabulary is dictionary of all the unique words from all the documents and each word gets unique index starting from 0.

lets look at the example where we have three sentences and each sentence represents a specific document :

- 'The sun is shining'
- 'The weather is sweet'
- 'The sun is shining, the weather is sweet, and one and one is two'

from our example we can create vocabulary dictionary as given below :

{'and': 0, 'two': 7, 'shining': 3, 'one': 2, 'sun': 4,
'weather': 8, 'the': 6, 'sweet': 5, 'is': 1}

here we can say 'and' is at index 0 and 'is' at index 1.

now we create a feature vectors which will look like this :

```
[[0 1 0 1 1 0 1 0 0],
[0 1 0 0 0 1 1 0 1],
[2 3 2 1 1 1 2 1 1]]
```

here each list is an individual document, in that list, for each index, we write how many times the word for that index, is appeared in that particular document. for example here in first list on first index we see how many times 'and' (because 'and' has index 0 in our vocabulary) appeared in that sentence which is 0 for next index we see how many times 'is' (because 'is' has index 1 in our vocabulary) appeared in that sentence and that is 1. in similar way for third sentence on index 0 we write 2 because 'and' is appeared 2 times in that particular sentence.

these values in feature vector are also called raw term frequencies: tf(t, d)—the number of times a term, t, occurs in a document, d.

> N in N-gram models represents the number of words we use as a single token our above bag-of-word model is 1-gram or unigram model.
> To summarize the concept of the n-gram representation, the 1-gram and 2-gram representations of our first document, “the sun is shining”, would be constructed as follows:
>
> - 1-gram: “the”, “sun”, “is”, “shining”
> - 2-gram: “the sun”, “sun is”, “is shining”

term frequency-inverse document frequency (tf-idf)

in the dataset there are some words that keep on repeating in each document, these words don't hold any important information, so we should be giving them less importance but in term frequencies we don't calculate based on this information and thats why we use this technique called **term frequency-inverse document frequency (tf-idf)** which can be used to downweight these frequently occurring words in the feature vectors.

below is the formula for tf-idf:

$$tf\text{-}idf(t,d) = tf(t,d) \times idf(t,d)$$

here idf(t,d) is **inverse document frequency**, where t is term and d is document, and we can calculate using given formula:

$$idf(t,d) = \log \frac{n_d}{1 + df(d,t)}$$

Here, nd is the total number of documents, and df(d, t) is the number of documents, d, that contain the term t. also adding one in denominator is optional but we do so, for the terms that occur in none of training examples so we could assign them non zero values.

these same formulae are applied in bit different way for scikit learn, as shown below :

$$idf(t,d) = \log \frac{1 + n_d}{1 + df(d,t)}$$

$$tf\text{-}idf(t,d) = tf(t,d) \times (idf(t,d) + 1)$$

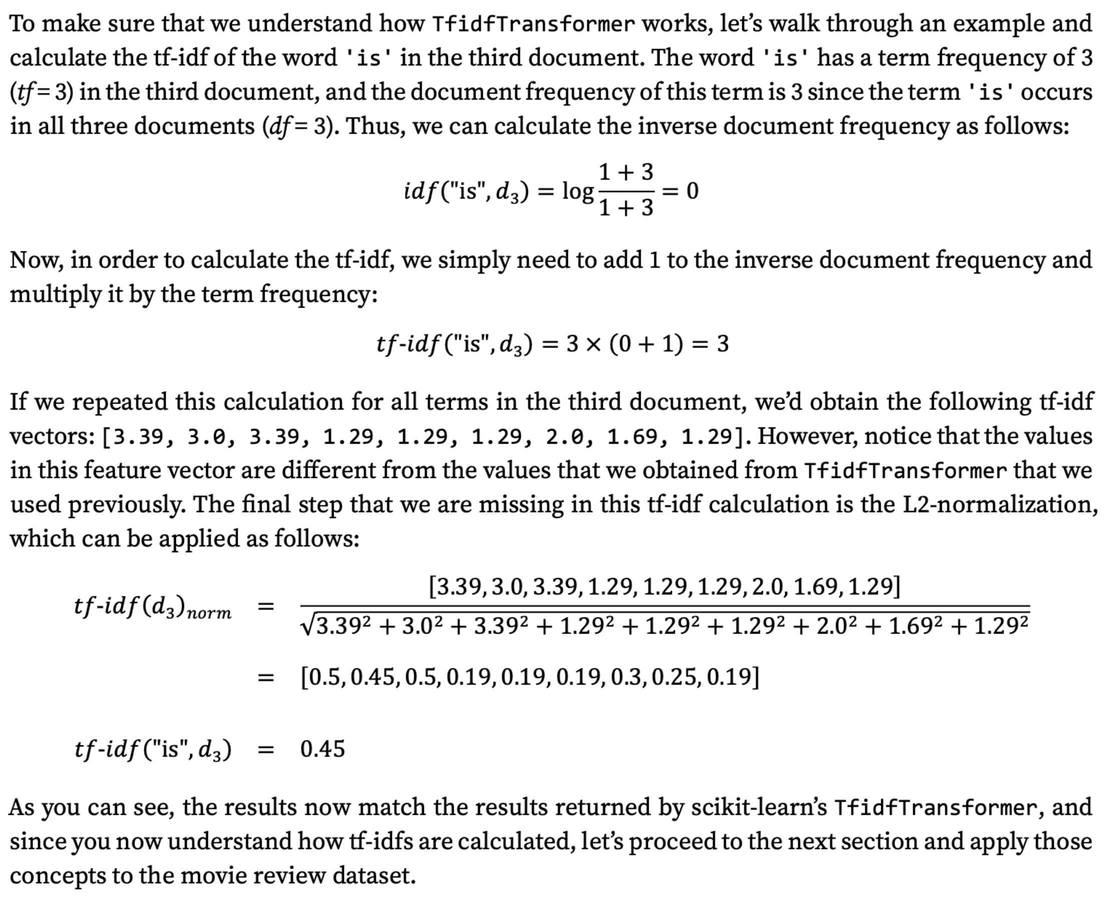
in first equation we do +1 in numerator, for assigning zero weight (that is, $idf(t,d) = log(1) = 0$) to terms that occur in all documents.
and in same way we change second equation to not make our term frequencies o in case of $idf(t,d) = 0$.

before returning feature vectors we can also normalize them, we can normalize them in following way using l2 normalization technique:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{v}{(\sum_{i=1}^{n} v_i^2)^{1/2}}$$

its just we device all the values in our vector with addition of squares of all the vectors and then taking their square root.

example for more clarity:



To make sure that we understand how `TfidfTransformer` works, let's walk through an example and calculate the tf-idf of the word `'is'` in the third document. The word `'is'` has a term frequency of 3 (*tf* = 3) in the third document, and the document frequency of this term is 3 since the term `'is'` occurs in all three documents (*df* = 3). Thus, we can calculate the inverse document frequency as follows:

$$idf(\text{"is"}, d_3) = \log\frac{1+3}{1+3} = 0$$

Now, in order to calculate the tf-idf, we simply need to add 1 to the inverse document frequency and multiply it by the term frequency:

$$tf\text{-}idf(\text{"is"}, d_3) = 3 \times (0+1) = 3$$

If we repeated this calculation for all terms in the third document, we'd obtain the following tf-idf vectors: `[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]`. However, notice that the values in this feature vector are different from the values that we obtained from `TfidfTransformer` that we used previously. The final step that we are missing in this tf-idf calculation is the L2-normalization, which can be applied as follows:

$$\begin{aligned} tf\text{-}idf(d_3)_{norm} &= \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}} \\ &= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19] \\ tf\text{-}idf(\text{"is"}, d_3) &= 0.45 \end{aligned}$$

As you can see, the results now match the results returned by scikit-learn's `TfidfTransformer`, and since you now understand how tf-idfs are calculated, let's proceed to the next section and apply those concepts to the movie review dataset.

word stemming and stop words

its the process of transforming words into their root form. here we use porter stemmer algorithm developed by Martin F. Porter.

for example:
normal sentence: `'runners like running and thus they run'`
after using porter stemmer algo: `['runner', 'like', 'run', 'and', 'thu', 'they', 'run']`

stop words:

these are the words that extremely common in language and hold no or very little useful information about the documents. Examples of stop words are is, and, has, and like.
when we work with raw term frequencies it is useful to remove these stop words but when we work with tf-idf they already downweight the frequently occurring words so we don't need it.

out-of-core learning

- in this we stream documents i.e we don't train entire model on all data at a time as its computationally expensive and can be time consuming.
- here we don't use tf-idf vectorizer as it requires all the dataset to be in memory, we use different kind of vectorizer called hashing-vectorizer which is data independent and uses hashing trick via the 32-bit MurmurHash3.
- so its just training a batch at a time instead of of all the examples

Topic modeling with latent Dirichlet allocation

- unsupervised -> clustering -> topic modeling
- we assign topic to each text document for example lets say we have lots of news article data, just a corpus of text, in topic modeling we assign topics to each document like sport, crime, politics etc.
- we will use famous technique know as latent dirichlet allocation in this section to do topic modeling

latent Dirichlet allocation

- in this technique we check for the same words that appear in multiple documents and then we cluster that documents into one topic.
- LDA is generative probabilistic model that tries to find group of words that appears frequently together across different documents. these frequently appearing words represent our topics.
- Given a bag-of-words matrix as input, LDA decomposes it into two new matrices:
    - A document-to-topic matrix
    - A word-to-topic matrix
- LDA decomposes the bag-of-words matrix in such a way that if we multiply those two matrices together, we will be able to reproduce the input, the bag-of-words matrix, with the lowest possible error.
- The only downside may be that we must define the number of topics beforehand—the number of topics is a hyperparameter of LDA that has to be specified manually.
- LDA gives us list of topics (number of topics are decide by us but not the topics itself) and each topic has a list of vector (the length of this vector is equal to the unique words

in our vocabulary) where each element in that vector represents the probability of that word being related to topic.

chapter 9 : Predicting Continuous Target Variables with Regression Analysis

`page no. 269 to 304`

colab notebook for all the code

overview

- linear regression : simple linear regression, multiple linear regression
- loading ames housing data into data frame with limited features
- visualizing data using variance techniques like graphs and correlation matrix
- implementing linear regression with mean squared error and gradient descent from scratch.
- implementation of linear regression in scikit-learn.
- implementation of linear regression using RANSAC.
- evaluation the performance of linear regression model.
  - residual plots of data
  - mean square error
  - mean absolute error
  - SSE and SST
- regularization for regression model
  - ridge regression
  - lasso
  - elastic net
- polynomial regression : for adding non-linearity.
- random forest regression for non-linear problems.

linear regression concept

the goal of linear regression model is to find relationship between one or multiple variables and a continuous variable (also know as target). main focus is our target variable is continuous.

simple linear regression :

- here we model the relationship between a single feature or variable (x) and continuous target variable (y).
- given is the equation for simple linear regression

$$y = w_1 + b$$

- we know here w is weight we want to learn and b is the bias unit and its also an y axis intercept.
- in the linear regression we try to find best fitting line for equation, this best fitting line is also called regression line, and the vertical lines from the regression line to the training examples are the so-called offsets or residuals—the errors of our prediction. given is the diagram to understand these concepts more throughly :

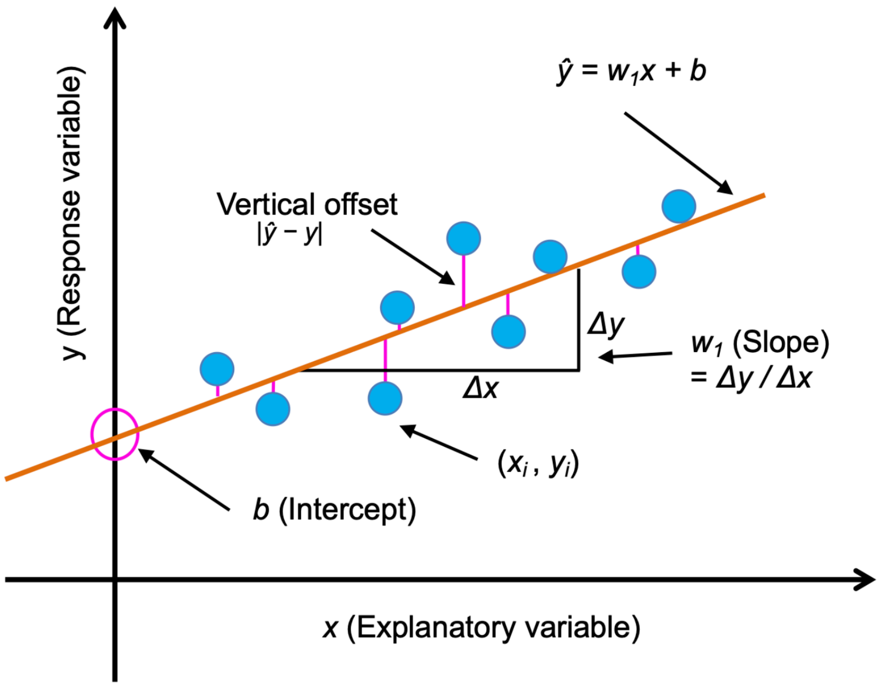


Figure 9.1: A simple one-feature linear regression example

multiple linear regression

- this is where we have multiple explanatory variables and one continuous target variable i.e more generalized form of simple linear regression.
- equation of multiple linear regression :

$$y = w_1x_1 + \cdots + w_mx_m + b = \sum_{i=1}^{m} w_ix_i + b = \mathbf{w}'\mathbf{x} + b$$

- given is diagram to understand the two-dimensional, fitted hyperplane of a multiple linear regression model with two features

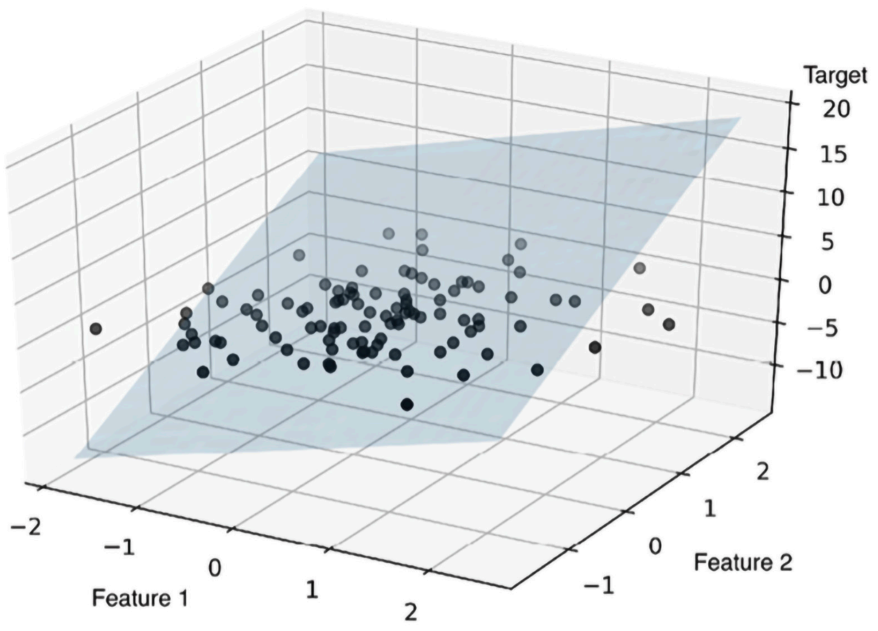


Figure 9.2: A two-feature linear regression model

-

EDA

- EDA : exploratory data analysis
- it means exploring our data and getting ourselves familiar with our data.
- using various methods like visualizing data using scatterplot

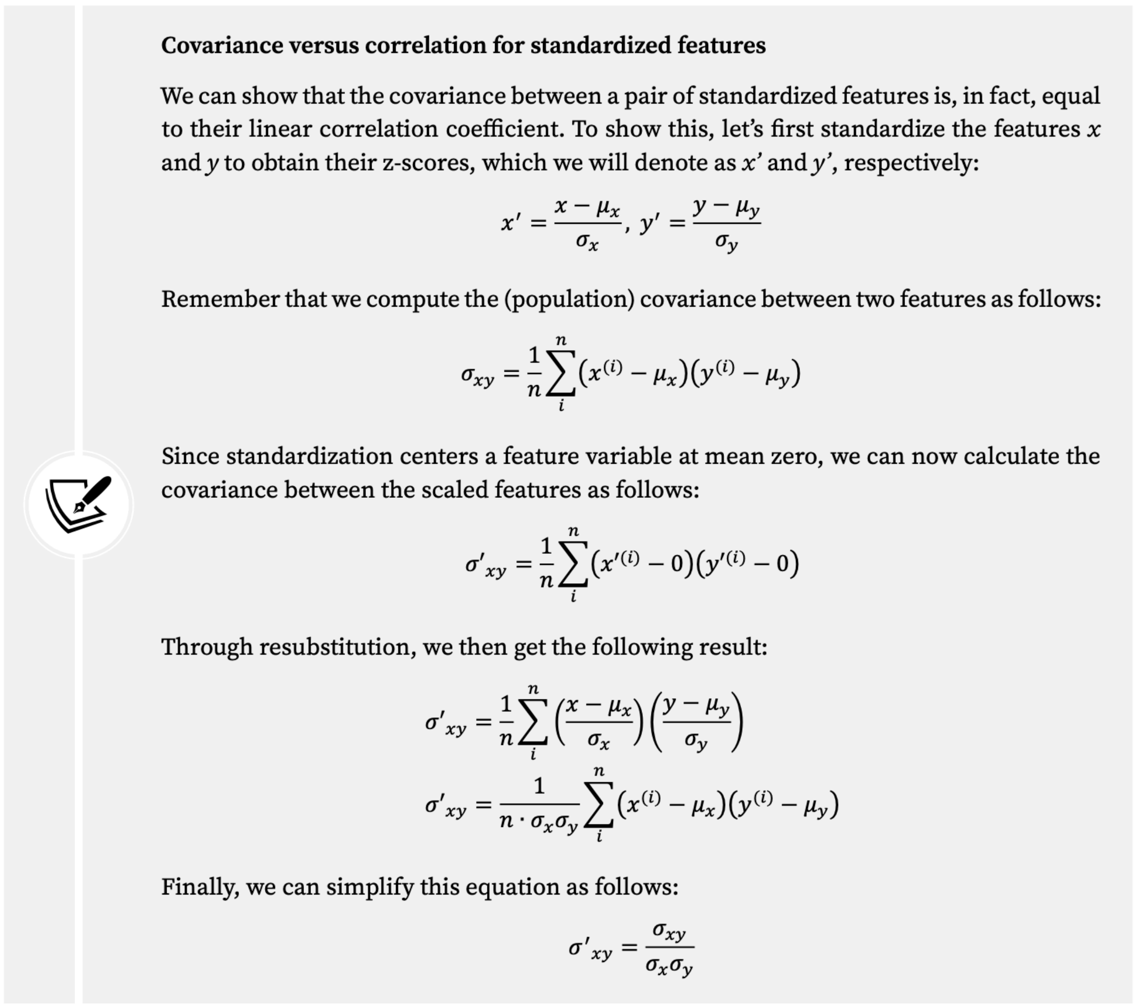
correlation matrix

The correlation matrix is a square matrix that contains the Pearson product-moment correlation coefficient (often abbreviated as Pearson's r), which measures the linear dependence between pairs of features. The correlation coefficients are in the range –1 to 1. Two features have a perfect positive correlation if r = 1, no correlation if r = 0, and a perfect negative correlation if r = –1.

$$r = \frac{\sum_{i=1}^{n}\left[(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)\right]}{\sqrt{\sum_{i=1}^{n}(x^{(i)} - \mu_x)^2}\sqrt{\sum_{i=1}^{n}(y^{(i)} - \mu_y)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

here μ represents the mean of current feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations.

> We can show that the covariance between a pair of standardized features is, in fact, equa to their linear correlation coefficient, given is the proof



Covariance versus correlation for standardized features

We can show that the covariance between a pair of standardized features is, in fact, equal to their linear correlation coefficient. To show this, let's first standardize the features x and y to obtain their z-scores, which we will denote as x' and y', respectively:

$$x' = \frac{x - \mu_x}{\sigma_x}, \ y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we compute the (population) covariance between two features as follows:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Since standardization centers a feature variable at mean zero, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x'^{(i)} - 0\right)\left(y'^{(i)} - 0\right)$$

Through resubstitution, we then get the following result:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\sigma'_{xy} = \frac{1}{n \cdot \sigma_x \sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Finally, we can simplify this equation as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

[for intuition of above formula click here](https://aistudio.google.com/app/prompts?state=%7B%22ids%22:%5B%2211jogAvtQBhExgRjvwcd-A1of4HggCLpM%22%5D,%22action%22:%22open%22,%22userId%22:%22109313882700383539784%22,%22resourceKeys%22:%7B%7D%7D&usp=sharing, https://drive.google.com/file/d/1c120G3XfyEnamIAgigvE_ZO9teDid5km/view?usp=sharing)

implementing linear regression

our own implementation in numpy

- here we will use OLS (ordinary least square) method to fit our model this method is also know as linear least square method.
- our loss function here is identical to MSE (mean square error), which is given by formula

$$L(w,b) = \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

- here $\hat{y}$ is the predicted value $\hat{y} = W^T x + b$, and we use term $\frac{1}{2}$ so it'll be easier to take its derivative in future.
- normal equation for linear regression i.e another way to do linear regression :

$$w = (X^T X)^{-1} X^T y$$

- above equation is guaranteed to give u most optimal solution but it is also very expensive computationally for large datasets hence we don't use it in real world applications.

linear regression in scikit-learn

- code for this is given in colab notebook :)

linear regression using RANSAC

RANdom SAmple Consensus (RANSAC) algorithm, this algorithm fits our model to subset of the data, which is called inliers.

general method of how RANSAC algorithm works :

1. Select a random number of examples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
3. Refit the model using all inliers.
4. Estimate the error of the fitted model versus the inliers.
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations was reached; go back to step 1 otherwise.

- we have this model in our scikit-learn too under `sklearn.linear_model` class.
- we use `residual_threshold=` in scikit-learn to set threshold for our model to detect outliers and inliers.
- when we set default value for threshold in scikit-learn, it uses the MAD estimate to select the inlier threshold, where MAD stands for the median absolute deviation of the target values, y.
- code snippet for MAD :

```
def median_absolute_deviation(data):
    return np.median(np.abs(data - np.median(data)))
```

evaluating regression model

residual plots

- residual : its the difference or vertical distance between predicted value and actual value.
- we can plot these residuals on graph and this graph is called residual plots, we plot these residuals against our predicted values.
- these residual plots are used to evaluate regression models as visualizing the multiple regression model is not possible.
- example :

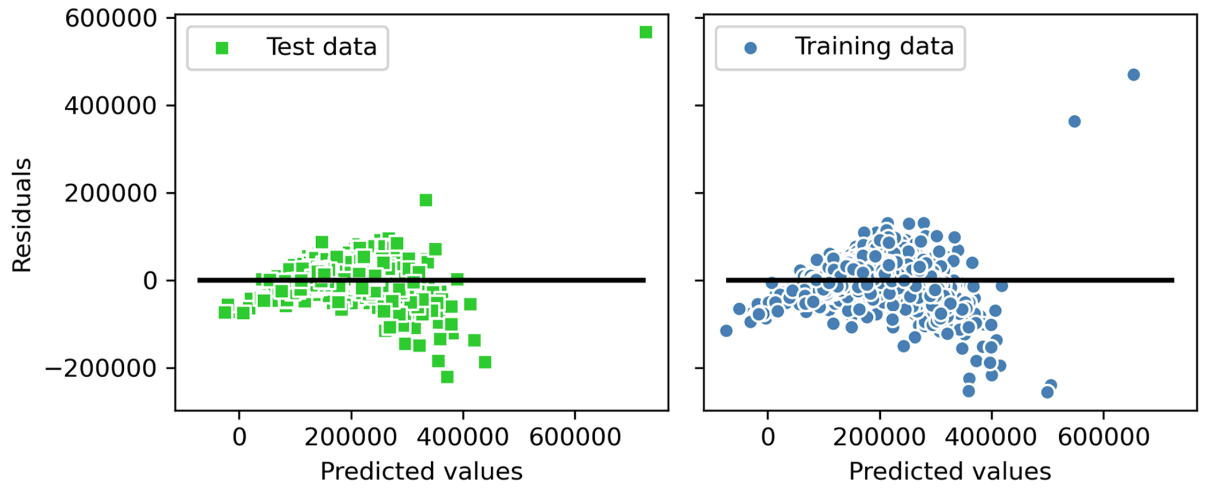


Figure 9.11: Residual plots of our data

- for our regression model to be perfect we need all residuals at zero, but its not possible for real world problems, for our model to be good it needs our residuals to randomly distributed along our x axis, if our residual plot shows some kinda of pattern, then that means our model is missing something important information regarding data, which is what happening in our above plot.
- Ideally, our model error should be random or unpredictable. In other words, the error of the predictions should not be related to any of the information contained in the explanatory variables; rather, it should reflect the randomness of the real-world distributions or patterns.

MAE (mean absolute error) and MSE (mean square error)

formula for MSE :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

we can use this MSE as scoring metric as we used accuracy in classification problem for cross-validation score.

sometimes its more intuitive to show error on actual scale, like in our case to show it in dollars and not in dollar squared and hence we take square root of MSE which is also know as MAE (mean absolute error).
given is the formulae for it :

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

MAE and MSE roles are scale dependent i.e $|\$500K - 550K| < |\$500{,}000 - 550{,}000|$

it may sometimes be more useful to report the coefficient of determination (R^2), which can be understood as a standardized version of the MSE, for better interpretability of the model's

performance.
R^2 is calculated as follows :

$$R^2 = 1 - \frac{SSE}{SST}$$

where SSE is the sum of square errors, which is similar to the MSE but does not include the normalization by sample size n:

$$SSE = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

and SST is variance of response;

$$SST = \sum_{i=1}^{n} \left(y^{(i)} - \mu_y\right)^2$$

extra equation for R^2 :

$$\begin{aligned} R^2 &= 1 - \frac{\frac{1}{n}\mathrm{SSE}}{\frac{1}{n}\mathrm{SST}} \\ &= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2} \\ &= 1 - \frac{\mathrm{MSE}}{\mathrm{Var}(y)} \end{aligned}$$

above equation shows us that R^2 is just scaled version of the MSE.

For the training dataset, R2 is bounded between 0 and 1, but it can become negative for the test dataset. A negative R2 means that the regression model fits the data worse than a horizontal line representing the sample mean. (In practice, this often happens in the case of extreme overfitting, or if we forget to scale the test set in the same manner we scaled the training set.) If R2 = 1, the model fits the data perfectly with a corresponding MSE = 0.

regularization in regression

ridge regression

this is a regression model where we add L2 regularization coefficient to penalize the wrong predictions given is the formula for it :

$$L(w)_{\text{Ridge}} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|w\|_2^2$$

and given is formula for L2 term :

$$\lambda \|w\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2$$

LASSO :

as we know in lasso regularization some weights can become zero which also makes it useful for supervised feature selection. given is the formula for lasso:

$$L(\mathbf{w})_{\text{Lasso}} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \|\mathbf{w}\|_1$$

and given is the formula for regularization term also know as penalty term:

$$\lambda \|\mathbf{w}\|_1 = \lambda \sum_{j=1}^{m} |w_j|$$

elastic net

A compromise between ridge regression and LASSO is elastic net, which has an L1 penalty to generate sparsity and an L2 penalty such that it can be used for selecting more than n features if m > n (where m is the number of features and n is the number of examples in training dataset):

$$L(\mathbf{w})_{\text{ElasticNet}} = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 + \lambda_1 \|\mathbf{w}\|_1 + \lambda_2 \|\mathbf{w}\|_2^2$$

non-linearity in regression

polynomial regression

when we encounter non-linear problems, we can add polynomial terms in our equation to deal with these non-linearity, even tho its polynomial regression its still considered as multiple regression model due to regression coefficients, w : (◕‿◕✿)💖

$$y = w_1 x + w_2 x^2 + \cdots + w_n x^d + b$$

Here, d denotes the degree of the polynomial.

decision tree regression

as we know in decision trees we try to find best feature to split the data on each node, we determine which feature is best using information gain i.e the feature that brings the best information gain is the best feature, given is the formula for information gain:
(maximizing IG is like finding the yummiest split that makes the purest baby nodes~ 🍬)

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

Here, x_i is the feature to perform the split, N_p is the number of training examples in the parent node,I is the impurity function, D_p is the subset of training examples at the parent node, and D_{left} and D_{right} are the subsets of training examples at the left and right child nodes after the split.

for classification tasks we usually use gini impurity or entropy as impurity metric, but for continuous variable we need different impurity metric, which is MSE: (◕‿◕✿)💕

$$I(t) = \text{MSE}(t) = \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{y}_t)^2$$

here N_t is the total number of example at that node t, D_t is the training subset at node t, $y^{(i)}$ is the true target value and $\hat{y}_t$ is the predicted value which can be calculated from given formula:

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

In the context of decision tree regression, the MSE is often referred to as *within-node variance*, which is why the splitting criterion is also better known as *variance reduction*.

random forest regression

as we know random forest is ensemble technique where we ensemble multiple decision trees, which gives us better results than individual tree due its randomness. this randomness helps to reduce model's variance. random forest is less sensitive to outliers and it also doesn't require much parameter tuning, the only parameter we have to tune is the number of decision trees.
difference between classification decision trees and regression difference is, that we use the MSE criterion to grow the individual decision trees, and the predicted target variable is calculated as the average prediction across all decision trees.

chapter 10 : Working with Unlabeled Data – Clustering Analysis

`page no. 305 to 334`

colab book with all the codes

overview

- k-means algorithm
  - math behind it
  - application in sklearn
  - smart way of clustering using k-means++ (for choosing better centre)
- hard vs soft clustering
- fuzzy C-means (FCM) algorithm
- evaluating clusters:
  - elbow method to find the optimal number of clusters
  - silhouette analysis
- hierarchical clustering

- agglomerative and divisive
- agglomerative
  - single linkage
  - complete linkage
- attaching our dendrograms (visual representation of data) to heatmaps
- DBSCAN algorithm: density-based spatial clustering of applications with noise algorithm

what is clustering?

- in unsupervised learning we don't have labels or answers or target variables.
- clustering is part unsupervised learning. its used to find hidden structures in data that we don't know. the goal of clustering is to find natural groups of data, such way that examples in same group are similar to each other than to those from different cluster.

k-means algorithm for clustering (prototype based clustering)

- we use this algorithm to find clusters in data i.e similar examples to each other are grouped together.
- k-means algorithm belongs to the category of prototype based clustering.
- this algorithm is very easy to implement plus its also computationally effective hence this is popular choice for clustering.
- one drawback for this algorithm is that we have to predefine the number of clusters we want to form i.e k.
- centroid : this is the average or mean of all data points in our cluster. it could be virtual point that does not exist in dataset.
- medoid : this is the point in dataset which is closest to all points i.e if we take the sum of all distances from each point in dataset then the one with least sum is medoid. it alway has to be real time.

stepwise algorithm

1. we choose the value for k i.e the number of cluster we want, then we randomly pick k centres from our dataset, for example if $k = 3$, pick random 3 points as centroid C_1, C_2, C_3.
2. now for each data point x_i :
   1. calculate the euclidian distance between datapoint and randomly chosen centroid C_k

$$\text{Distance} = \sqrt{(x_i - C_k)^2} \quad (\text{for each centroid } k)$$

   2. Assign xixi to the cluster with the closest centroid.
   - as a result of this process now we will have k clusters
3. here we recalculate the centroids;

    1. For each cluster, compute a new centroid as the mean (average) of all points in that cluster
  - we do this process so that we can get the new centroid for each cluster and this centroid is at the centre of our cluster.
4. we repeat the step 2 n 3 until;
  - centroid stabilize
  - cluster assignment stop changing
  - a maximum number of iterations is reached
  - once one of three given conditions are satisfied we stop the algorithm and return k clusters.

we can also describe k-means algorithm as simple optimization problem, an iterative approach for minimizing the within-cluster sum of squared errors (SSE), which is sometimes also called cluster inertia:

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \| x^{(i)} - \mu^{(j)} \|_2^2$$

Here, $\mu^{(j)}$ is the representative point (centroid) for cluster j. $w^{(i,j)} = 1$ if the example, $x^{(i)}$, is in cluster, or 0 otherwise.

$$w^{(i,j)} = \begin{cases} 1, & \text{if } x^{(i)} \in \text{cluster } j \\ 0, & \text{otherwise} \end{cases}$$

- in k-means algorithm there exist a problem, which is that sometimes a cluster can be empty in such a case, the algorithm will search for the example that is farthest away from the centroid of the empty cluster. Then, it will reassign the centroid to be this farthest point.
- also feature scaling is important for k-means algorithm as we use euclidian distance as metric.
- this algo also has the problem that we assume our clusters to be in circular shape, if our clusters are of some other complex shapes then this algo fails.

improved version of k-means : k-means++

- as we know k-means algo initializes centroids randomly, sometimes they can be poorly initialized causing slow convergence or bad clustering. this newer version of k-means deals with this by initializing centroids far away from each other.

given is stepwise algorithm:

1. Initialize an empty set, M, to store the k centroids being selected.
2. Randomly choose the first centroid, $\mu^{(j)}$, from the input examples and assign it to M.
3. For each example, $x^{(i)}$, that is not in M, find the minimum squared distance, $d(x^{(i)}, M)^2$, to any of the centroids in M. given is the formula for finding squared distance:

$$d(x,y)^2 = \sum_{j=1}^{m}(x_j - y_j)^2 = \|x-y\|_2^2$$

4. To randomly select the next centroid, $\mu^{(p)}$, use a weighted probability distribution equal to:

$$\frac{d(\mu^{(p)}, M)^2}{\sum_i d(x^{(i)}, M)^2}$$

For instance, we collect all points in an array and choose a weighted random sampling, such that the larger the squared distance, the more likely a point gets chosen as the centroid.
5. Repeat steps 3 and 4 until k centroids are chosen.
6. Proceed with the classic k-means algorithm.

tldr: we choose first centroid randomly and then add it to our list, then for all the examples that are not in the list we find euclidian distance between centroids and other datapoints, then we choose the next centroid which has the most euclidian from our centroids, but we do this using weighted probability distribution, and this way we get centroids that are far from each other.

FCM (fuzzy C-means) algorithm

hard vs soft clustering

- in hard clustering each example is assigned to one cluster only but in soft clustering each example can be assigned to one or more clusters.
- soft clustering is also known as fuzzy clustering.

FCM

- fuzzy C-means algorithm is also know as soft k-means and fuzzy k-means algorithm
- the main difference between this algorithm and k-means is instead of assigning each example to one cluster, for each example we assign probability for each cluster.

for example: in k-means our example will be associated with only one cluster and that can be shown in following way,

$$\begin{bmatrix} x \in \mu^{(1)} & \rightarrow & w^{(i,j)} = 0 \\ x \in \mu^{(2)} & \rightarrow & w^{(i,j)} = 1 \\ x \in \mu^{(3)} & \rightarrow & w^{(i,j)} = 0 \end{bmatrix}$$

here our example x is assigned to cluster $\mu^{(2)}$ and thats why $w^{(ij)}$ for it is 1.

in case of fcm we will assign probability for each cluster as given,

$$\begin{bmatrix} x \in \mu^{(1)} & \rightarrow & w^{(i,j)} = 0.1 \\ x \in \mu^{(2)} & \rightarrow & w^{(i,j)} = 0.85 \\ x \in \mu^{(3)} & \rightarrow & w^{(i,j)} = 0.05 \end{bmatrix}$$

as we can see for example x we are assigning probabilities for each cluster μ.

- as these values are probabilities their sum should be equal to 1.

stepwise algorithm :

1. Specify the number of k centroids and randomly assign the cluster memberships for each point.
2. compute the cluster centroids:
    - for each cluster we find weighted average of all examples, where weight is probability of each example associated with that cluster, given is the formula for it;

$$\mathbf{v}_i = \frac{\sum_{j=1}^{N} (\mu_{ij})^m \cdot \mathbf{x}_j}{\sum_{j=1}^{N} (\mu_{ij})^m}$$

- so we are finding centroid for ith cluster, and hence in above equation μ_{ij} represents, weight of jth example for ith cluster, and m is fuzziness exponent, and X_j is jth datapoint or example.
3. update the weights (i.e probabilities associated with each cluster for each example):
- we use the following formula to update our weights:

$$\mu_{ij} = \frac{1}{\sum_{k=1}^{C} \left(\frac{\|x_j - v_i\|}{\|x_j - v_k\|} \right)^{\frac{2}{m-1}}}$$

- in this step we find new weights for each example and for each cluster (our each example have multiple weights and each weight term is probability for a particular cluster).
- now in above equation u_{ij} is weight associated with x_jth example and ith cluster, on right side equation v_i is the centroid for ith cluster that we calculated in previous step, v_k is the centroid for kth cluster.
- intuition is when our example or datapoint is closer to ith centroid the term $\|x_j - v_i\|$ will be smaller than $\|x_j - v_k\|$ causing the entire denominator to be small which makes μ_{ij} to be bigger. to summerize whole point when our datapoint is closer to the centroid it will increase the weight associated with that cluster.
- when data point is further than its current centroid (current here means v_i which is centroid of cluster i) opposite happens of what happened above i.e denominator gets bigger and causes μ_{ij} to be small.

4. Repeat steps 2 and 3 until the membership coefficients do not change or a user-defined tolerance or maximum number of iterations is reached

fuzziness coefficient:

The fuzziness coefficient (denoted as m) is a critical parameter in the Fuzzy C-Means (FCM) algorithm that controls how "soft" or "strict" the cluster assignments are.

- Purpose: Governs the degree of overlap between clusters.
  - Higher mm → More overlap (fuzzier boundaries).
  - Lower mm → Less overlap (crisper boundaries, closer to K-means).
- Key Insight:
  - The exponent $\frac{2}{m-1}$ determines how sharply memberships drop with distance.
  - Higher m → Smaller exponent → Memberships decay more slowly (points belong to multiple clusters more evenly).
  - Lower m → Larger exponent → Memberships decay rapidly (points strongly favor the nearest cluster).

evaluation of clustering algorithms

- as we don't have class labels in clustering so we depends on intrinsic metrics—such as the within-cluster SSE (distortion)—to compare the performance of different k-means clustering models.

elbow method

its a method used to find optimal number of clusters (K) using value of SSE, as we increase the value of K SSE decreases (this is what our goal is when training model), but to many K can make our model overfit, so we need a point where K such that if we decrease the value K than this, the results worsens exponentially i.e SSE increases by a lot. this phenomenon creates graph that looks like elbow hence this method is called elbow method.

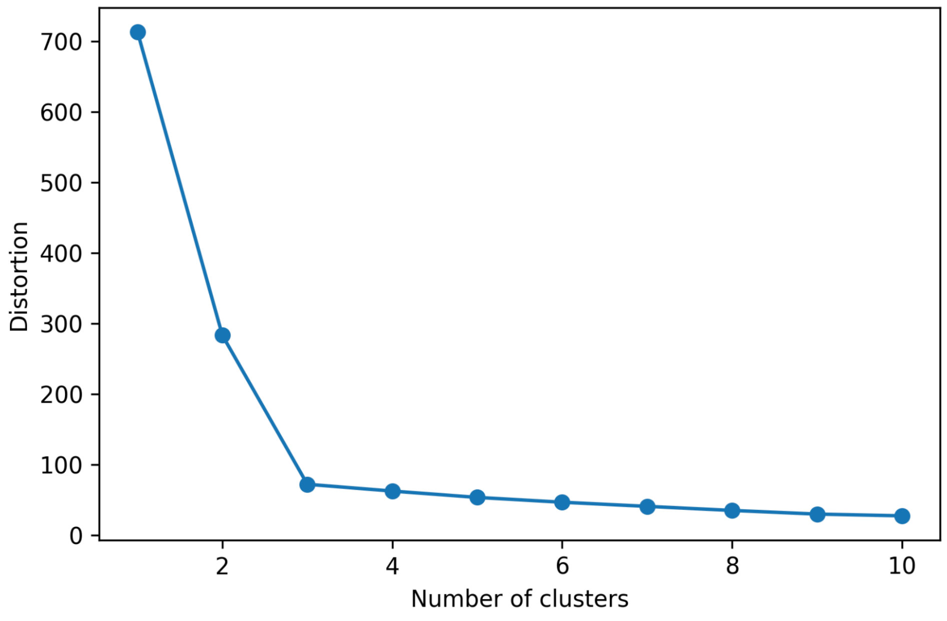


Figure 10.3: Finding the optimal number of clusters using the elbow method

from this diagram we can see that 3 is elbow point.

silhouette analysis

1. Calculate the *cluster cohesion*, $a^{(i)}$, as the average distance between an example, $x^{(i)}$, and all other points in the same cluster.
2. Calculate the *cluster separation*, $b^{(i)}$, from the next closest cluster as the average distance between the example, $x^{(i)}$, and all examples in the nearest cluster.
3. Calculate the silhouette, $s^{(i)}$, as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}$$

- bove equation $b^{(i)}$ quantifies how dissimilar an example is from other clusters, and $a^{(i)}$ tells us how similar it is to the other examples in its own cluster.
- if the silhouette coefficients are not close to 0 and are approximately equally far away from the average silhouette score, then its an indicator of good clustering.
- when silhouette have visibly different lengths and widths, its considered as bad clustering.

hierarchical clustering

- dendograms: visual representation of binary hierarchical clustering.

2 types of hierarchical clustering

divisive: in this type of clustering we start with one cluster that consists the entire dataset, and we iteratively split the cluster into smaller clusters until each cluster contains only one example.

agglomerative: here we start with each example as an individual cluster and merge the closest pairs of clusters until only one cluster remains. it has two types simple linkage and complete linkage which we will discuss in next section.

simple n complete linkage

- **simple linkage**:
  - in this method we find the distance between most similar member (these are two different datapoints from two different clusters that have the least distance between them) for each pair of clusters and merge the two clusters for which the distance between most similar members is the smallest.
- **complete linkage**:
  - this is similar to simple linkage but only difference is, instead of using most similar members we use most dissimilar members and merge the two clusters for which the distance between most dissimilar members is the smallest.

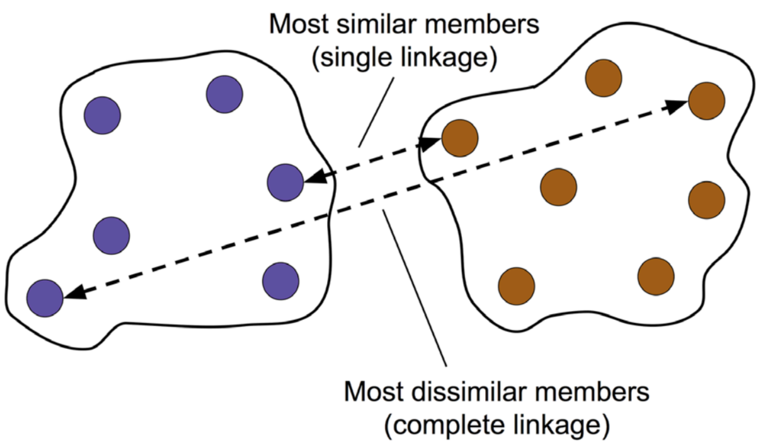


Figure 10.7: The complete linkage approach

hierarchical complete linkage clustering / agglomerative complete linkage

- **process/steps**
  1. Compute a pair-wise distance matrix of all examples (we can use euclidean distance as metric).
  2. Represent each data point as a singleton cluster.
  3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members.
  4. Update the cluster linkage matrix.
  5. Repeat steps 2-4 until one single cluster remains.

DBSCAN (density based spatial clustering of applications with noise)

- in DBSCAN the notion of density is defined as number of points within specified radius.
- a DBSCAN gives label to each point in dataset and the rules of labelling are as given :
  - if a point has specified number (MinPts) of neighbours, within a specified radius ϵ then this point is called **core point**.
  - if a point does not have MinPts within specified radius but it does comes in a range some other core point then its called **border point**.
  - all other points that are neither of those are considered as **noise points**.
- now that we have these labels we can write this algorithm in 2 simple steps:
  - Form a separate cluster for each core point or connected group of core points. (Core points are connected if they are no further away than ϵ.
  - Assign each border point to the cluster of its corresponding core point.

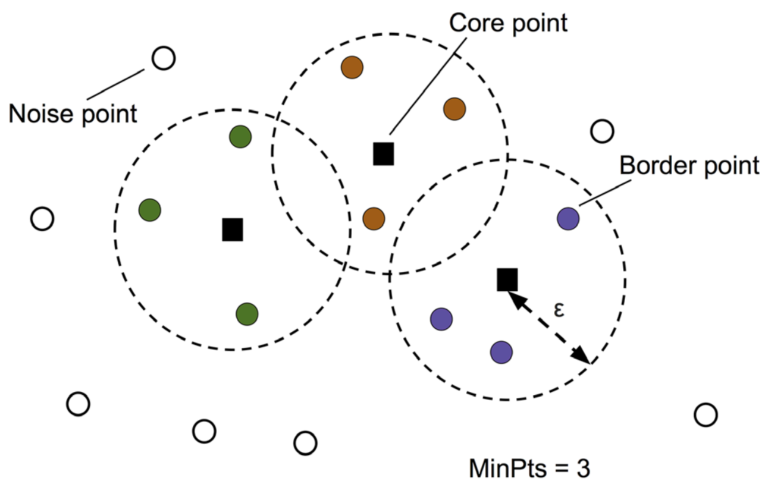


Figure 10.13: Core, noise, and border points for DBSCAN

- DBSCAN is better than both k-means and agglomerative algorithms when it comes to clustering complex shapes like half moons.

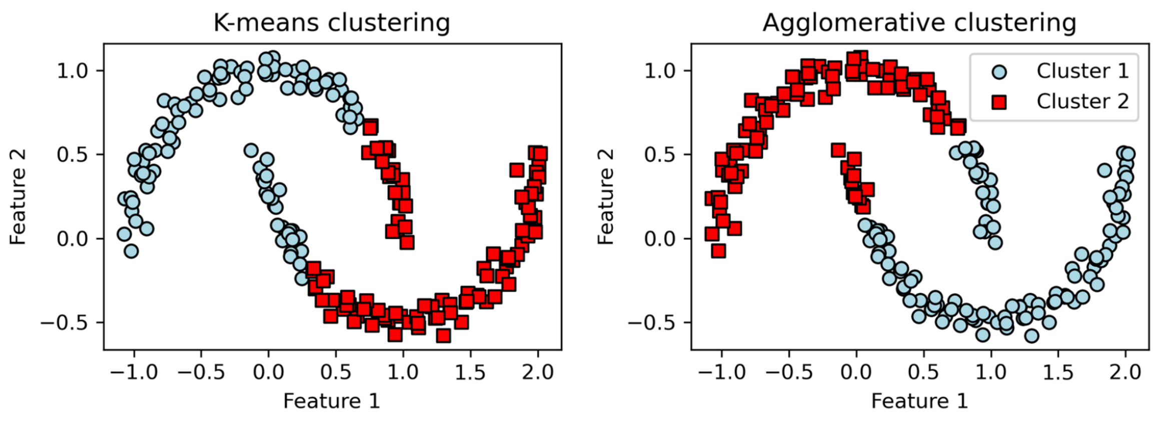


Figure 10.15: k-means and agglomerative clustering on the half-moon-shaped dataset

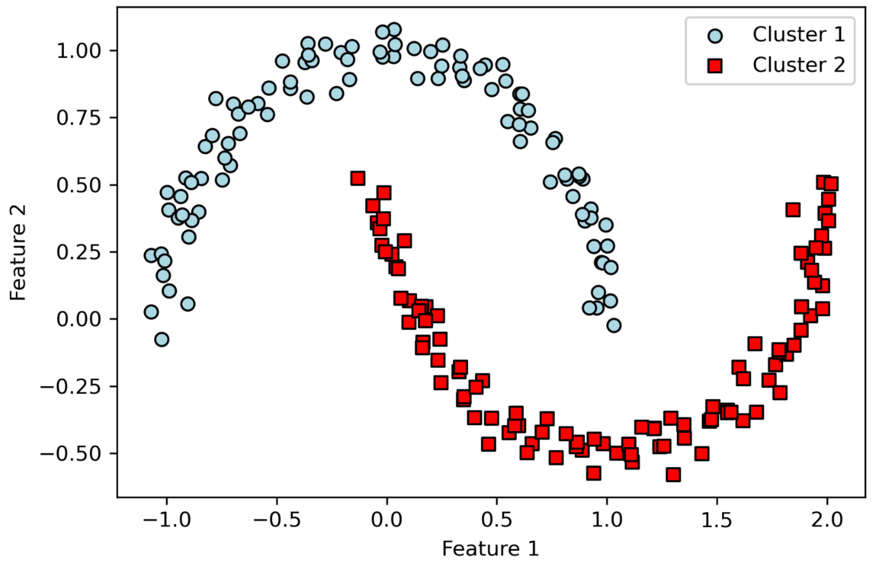


Figure 10.16: DBSCAN clustering on the half-moon-shaped dataset

- DBSCAN has also its advantages like **curse of dimensionality**, and its also more expensive to finetune as it have more hyperparameters. curse of dimensionality also effects k-means and hierarchical (agglomerative) algorithms as we use euclidean distance in all of them.

ending

Note that, in practice, it is not always obvious which clustering algorithm will perform best on a given dataset, especially if the data comes in multiple dimensions that make it hard or impossible to visualize. Furthermore, it is important to emphasize that a successful clustering does not only depend on the algorithm and its hyperparameters; rather, the choice of an appropriate distance metric and the use of domain knowledge that can help to guide the experimental setup can be even more important.

end of classical ml