BAYESIAN THEOREM

* $P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$ → class prior; → likelihood; → predictor prior

$P(H|E)$: posterior

* It takes the data calculates probabilities of each E_i seperately for all classes.
  - Normal mail : $P(E_1), P(E_2) \dots \rightarrow P(E_1|N), P(E_2|N)$
  - Spam mail : $P(E_1), P(E_2) \dots \rightarrow P(E_1|S), P(E_2|S)$

* It calculates a priori probabilities of $P(N), P(S)$ by simple proportion calculation.

* Then, calculate $P(N|E)$ etc.. is calculated from the apriori probability and conditional probabilit

Naive: It assumes that evidence splits into parts that are conditionally independent!

Zero Frequency Problem: If one of the attributes has never occured in one class, its prob $P(Ex|S)$ will be 0. So, it will disrupt.

⇒ ADD α (Laplace est.) ~ 1 (usually) for each attribute in each class, and calculate probabilities.

* Generally used in:
  - Real Time Prediction
  - Multiclass Prediction
  - Text classification / sentiment Analysis
  - Recommendation System with collaborative filtering

NOTES OF USAGE

→ Don't forget about ZERO FREQUENCY problem.

→ Independence of attributes is a strong assumption, beware!

→ It is fast & successful when independence holds.

→ Its probability might be meaningless, don't use it.

* If the data is categorical → OK!

If not: **Gaussian Naive Bayes**

- which assumes variables follow a normal distribution (STRONG ASSUMPTION)

PYTHON

* sklearn.naive_bayes

- GaussianNB() → non-categorical variables
- MultinomialNB() → categorical variables
- ComplementNB() → imbalanced datasets (adaptation of M. NB)
- BernoulliNB() → work with already binary or binarize data → non occurence counted
- CategoricalNB() → categorically distributed data

- .fit(X_train, Y_train)
- .predict(X_test)

} for all of them...

Binomial Distribution

* **Bernoulli**: A random experiment with exactly two possible outcomes → Fail / Success

* **Binomial**: Number of successes for a sequence of Bernoulli trials.

* $f(x) = \binom{n}{x} p^x (1-p)^{n-x}$

  → Fixed # of trials.
  → Random # of successes.

- $\mu = n \cdot p$
- $\sigma^2 = n \cdot p(1-p)$

Geometric Distribution

→ Fixed # of successes
→ Random number of trials

} For x success, how many trials should be done?

* $f(x) = p(1-p)^{x-1}$
  ↳ where x is the number of failures until the 1st success

- $\mu = \frac{1}{p}$
- $\sigma^2 = \frac{(1-p)}{p^2}$

→ **Lacks of Memory**: So if we start trial number i in the process, it does not differ from starting at 0.

Poisson Distribution

→ Expresses the probability of a given number of events occuring in a fixed interval of time / space.

* $f(x) = \frac{e^{-\lambda} \cdot \lambda^x}{x!}$ (λ is the rate per time/space)

  * λ should be events per unit time!
  ↳ if not, convert it!

- $\mu = \lambda$
- $\sigma^2 = \lambda$

Poisson Approximation to Binomial

* Binomial cases when
  - n is large ↑
  - p is small ↓

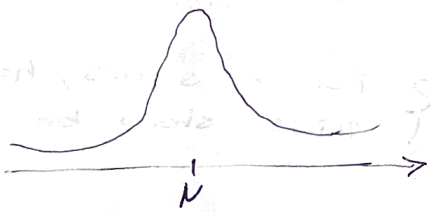
  } $np < 10$

THEN:

* Use Poisson with $\lambda = np$

CONTINUOUS

Normal Distribution



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

- $E(x) = N$
- $V(x) = \sigma^2$

} $N(N, \sigma^2)$

→ Standart z-table gives cumulative prob. distribution for certain values in z.

APPROX. TO BINOMIAL

$P(X \leq A) \approx P(X \leq A + 0.5)$

↳ where $N = np$

$\sigma^2 = np(1-p)$

↳ use normal z conversion → NORMAL PROBABILITY

APPROX. TO POISSON

$P(X \leq 950) = \sum_{0}^{950} \frac{e^{-1000} \cdot 1000^x}{x!}$ WUFF!

$\approx P(X \leq 950.5)$ and $N = 1000$ (λ), $\sigma^2 = 1000$ (λ) → NORMAL z.

Exponential Distribution

→ On probabilities of events occuring at a time, or in a range of time.

* $f(x;\lambda) = \begin{cases} \lambda \cdot e^{-\lambda x}, & x \geq 0 \\ 0, & x < 0 \end{cases}$

* $P(X \leq x) = 1 - e^{-\lambda x}$

  • $\mu = \frac{1}{\lambda}$

  • $\sigma^2 = \frac{1}{\lambda^2}$

→ Also has lack of memory property.

Beta Distribution (α, β)

→ A probability distribution on probabilities. $[0, 1]$

→ Binomial models number of successes

whereas

Beta models the probability of success

* $\alpha - 1$ → # of successes
* $\beta - 1$ → # of failures

Gamma Distribution (k, θ)

→ Predicts the wait time until the k^{th} event occurs (whereas exponential predicts the wait time for 1st event

k = # of events (scale) } Positive
θ = shape - rate of events }

* $k = 1$ → EXPONENTIAL
* k = integer → ERLANG

STATISTICS

* Frequentists define probability as the

  (*) Long-term frequency in a repeatable random experiment

* They do not use probability to quantify incomplete knowledge

  (*) measure degree of belief in hypothesis

HYPOTHESIS TESTING

We propose:

H_0 = Null Hypo

H_A = Alternative Hypo

⇒ We try to reject H_0 in the direction of H_A so that we can drive conclusions.

- One-Sided $p = P(Z > z | H_0)$
- Two Sided $p = P(|Z| > z | H_0)$

} p-values

p-value: is the probability that of the values farther to the center than this value.

Ex: A p-value of 0.05 means that there is 5% chance of finding more deviated points.

(GIVEN THAT H_0 !)

* Whilst doing that, p-value is the probability given that H_0 is true. If $H_0 = N=0$:

  $p = 0.32$ is the test statistic being the closest among 32% of the data if the center is 0.

  ↳ In this case, data suggests that not significantly away from assumed center, not at the tails → So, N can be 0. Cannot reject!

ERRORS & MEANINGS

	REALITY: H_0	REALITY: H_A
Reject H_0	Type I	✓
Don't Reject H_0	✓	Type II

* α = Significance = P(Type I) } ~ 0 ✓
  * probability we incorrectly reject H_0.
* β = power = 1 − P(Type II) } ~ 1 ✓
  * probability we correctly reject H_0.

Student t-test

→ When variance is unknown, we have to do an estimation.

$$s^2 = \frac{\sum_i (x_i - \bar{x})^2}{N-1}$$

→ N-1 since $\bar{x}$ is an estimation as well.

→ And the t-test becomes:

$$T = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

Chi-square Distribution

$$\sum_i \left(\frac{x_i - \mu_i}{\sigma_i}\right)^2 = \chi^2 = \frac{(n-1)\cdot s^2}{\sigma^2} \quad (n-1 \text{ d.of.})$$

STATISTICAL METHODS

SAMPLING

↳ Random Sampling
↳ Systematic Sampling
↳ Stratified Sampling ...

RESAMPLING

Once we sample the data, we may try to make a statistical inference about the population from a small set of observations.

↳ But with one sample we have a single estimate with little idea of the variability in the estimate.

* These comes resampling!

* NORMAL SUB-SAMPLES ~ Regular // without replacement //

BOOTSTRAP

→ Sampling with replacement
→ There are Out-of-Bag (OOB) data for testing / validating purposes.
→ Might be used for model development, confidence interval building, estimation etc.

* (Sample size might be equal to the whole data)

PYTHON

- sklearn.utils

  L resample (data,
  replace = True,
  n_samples = ,
  random_state =)

METRICS

Confusion Matrix

Predicted \ Actual	Positive	Negative
Positive	TP ✓	FP ✗
Negative	FN ✗	TN ✓

↳ Binary classification matrix

Accuracy

* $\frac{TP + TN}{TP + TN + FN + FP}$ → How accurate I am?

⚠ Not good in imbalanced data ⚠

Precision

* $\frac{TP}{TP + FP}$ → How precise I am when I guess positive?

Recall

* $\frac{TP}{TP + FN}$ → How succesfully recall the positives when I am guessing?

F1 Score

* $2 \cdot \frac{1}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}$

→ Harmonic mean between precision & recall.

→ How precise your classifier is as well as how robust it is (not missing significant number of instances)

⇒ Here, we are more focused on the positive class where we try to be precise in our predictions and also capture as much as the positive class at the same time.

Log Loss ~ cross-entropy function (used as metric as well)

→ works by penalizing the false classifications.

→ Works well for MULTI-CLASS classification.

→ Classifier should assign probabilities to each class for all sample.

* N samples belonging to M classes :

$$\text{LogLoss} = \frac{-1}{N} \sum_{i}^{N} \sum_{j}^{M} y_{ij} * \log(p_{ij})$$

↳ [illegible]

↳ When we make it for binary prediction : (M=2)

$$\text{Logloss} = \frac{-1}{N} \sum_{i=1}^{N} [\, y_i . \log(p_i) + (1-y_i) . \log(1-p_i) \,]$$

* It actually can be driven from maximum likelihood of bernoulli trial.

* Since it penalizes distances from the actual values (in prob. of 0-1) it turns out to be a negative value but should be used as a minimization problem

↳ THEREFORE NEGATIVE SIGN.

AREA UNDER CURVE

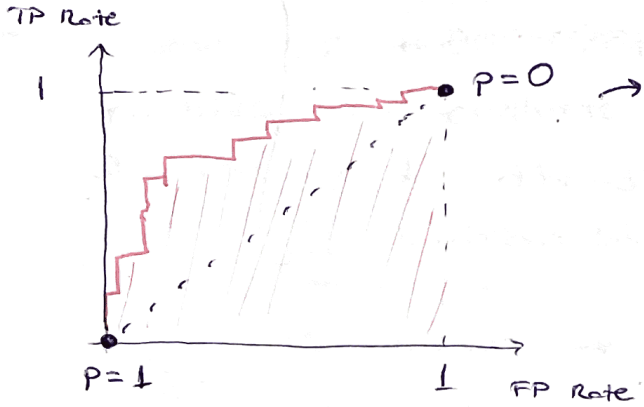
→ Known as C-value, as well.

→ It is the probability that when one sample from positive class and one from negative class are drawn, the prob. assigned to positive is greater than the negative (ranked higher). → RANKING ORDER...

→ TP (Sensitivity) Rate and FP (1-specificity) Rate is compared.

* **TP Rate** : $\frac{TP}{TP+FN}$ ~ How much of the actual positive I guessed/recalled

* **FP Rate** : $\frac{FP}{FP+TN}$ ~ How much of actual negative I guessed INCORRECTLY positive.



→ **Receiver Operating Curve**

p = threshold whose above is predicted 1 (positive)

* Drawn for all p values (thr) rates are calculated ...

↳ Area under Curve (AUC) is the metric.

Continuous

Mean Absolute Error

$$MAE = \frac{1}{N} \sum_j |y_j - \hat{y}_j|$$

↳ how far the predictions from the actual output.

Mean Squared Error

$$MSE = \frac{1}{N} \sum_j (y_j - \hat{y}_j)^2$$

R^2 & Adjusted R^2 ~ coefficient of determination

* R^2 is a metric based on how much of the variance in the data is explained by the suggested model.

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

$$\text{Adj. } R^2 = 1 - \frac{(n-1)}{[n-(k+1)]} \cdot (1 - R^2)$$

↳ penalizes addition of useless variables...

PYTHON IMPLEMENTATION

* sklearn.metrics

- .accuracy_score()
- .f1_score()
- .log_loss() ~ need predict_proba
- .precision_score()
- .recall_score()
- .roc_auc_score() ~ need predict_proba

} CLASSIFICATION

- .explained_variance_score()
- .max_error()
- .mean_absolute_error()
- .mean_squared_error()
- .median_absolute_error()
- .r2_score()
- .mean_absolute_percentage_error()

} REGRESSION

~~## N[illegible] CROSS VALIDATION~~

- sklearn.model_selection.cross_val_score

```
( classifier_object,
    X,
    Y,
    cv= k (fold),
  scoring = 'name of scoring method' )
```

0 sklearn.metrics.SCORERS is a dict
↳ .keys() gives allowed metric names.

- sklearn.metrics.classification_report(y_true, y_pred, target_names=[])
  ↳ gives precision/recall/f1 for each class.

LOSS FUNCTIONS

* Functions that are defined as the deviation (somehow) of predictions from actual values.
* Many (almost all) algorithms use one and improve itself by trying minimizing the loss function.
* Some of them (maybe all) can be used as metrics to evaluate the model performance in the end through VALIDATION data.

⇒ COST FUNCTION

- Loss function → For single training example
- COST function → Average loss for entire dataset.

LOSS

CLASSIFICATION	MULTI-CLASS	REGRESSION
- Binary Cross Entropy (Log-Loss)	- Multiclass Cross Entropy	- Squared Error
- Hinge Loss (For labels -1 & 1)	- Kullback-Liebler Divergence	- Absolute Error
		- Huber Loss

CLASSIFICATION

Binary Cross Entropy (Log-Loss)

$$L = -y \cdot \log(p) - (1-y)\log(1-p) = \begin{cases} -\log(1-p) & \text{if } y=0 \\ -\log(p) & \text{if } y=1 \end{cases}$$

⇒ For calculating p:

SIGMOID - $\frac{1}{1+e^{-z}}$ where z: a function of our input features

Hinge Loss

* Primarily used with SVM with classes [-1, +1]

$$L = MAX(0, 1 - y * f(x)) \Rightarrow \text{for } y \in [-1, 1]$$

* Not only penalizes wrong predictions but also the right predictions that are not confident.

* Used when we want to make real-time decisions with not a laser-sharp focus on accuracy.

MULTICLASS CLASSIFICATION

Multi class Cross-Entropy function

→ A generalization of the Binary Cross Entropy Loss.

- $L(x_i, y_i) = -\sum_{j=1}^{C} y_{ij} \cdot \log(p_{ij})$

↳ where y_i one-hot encoded vector

SOFTMAX : $\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$

* is implemented through a neural network layer just before the output layer.

↳ The softmax layer must have the same number of nodes as the output layer.

Kullback - Liebler Divergence

→ A measure of how a probability distribution differs from another distribution. If zero → identical distributions

$D_{KL}(P||Q) = \int P(x) \cdot \log\left(\frac{P(x)}{Q(x)}\right) \cdot dx$ (or $\sum$ for discrete)

↳ Expectation of logarithmic difference between P & Q in respect to P.

→ Not SYMMETRIC. So cannot be used as a distance metric.

① Minimize forwards KL → $D_{KL}(P||Q)$ → Supervised

② Minimize backward KL → $D_{KL}(Q||P)$ → Reinforcement

→ $D_{KL}(P||Q) = -\sum_x (P(x) \cdot \log(Q(x)) - P(x) \cdot \log(P(x))) = H(P,Q) - H(P,P)$

where $H(P,Q)$ is the cross-entropy of P & Q

$H(P,P)$ is the entropy of P.

REGRESSION

Squared Loss Error

$L = (y - f(x))^2$ → cost function MSE

Absolute Error Loss

$L = |y - f(x)|$ → cost function MAE

Huber Loss

→ Combines the best of MSE and MAE.

→ Quadratic for smaller errors and is linear otherwise.

$$L_\delta = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \le \delta \quad \} \text{ Quadratic} \\ \delta |y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \quad \} \text{ Linear} \end{cases}$$

⇒ Basically select δ and if Abs. Error is smaller then take Squared Error, else a function of Abs. Error & δ.

→ Used in Robust Regression / M-Estimation / Additive Modelling

- tensorflow.keras.losses

OPTIMIZATION ALGORITHMS

* Tries to find input parameters, weights etc. to a function that result in the minimum / maximum output of the function.

* With loss functions, ML algorithms use optimization algo's to minimize losses and find an optimal model.
  ↳ Not specifically loss but COST function for the model. But COST function is an aggregation of the LOSS function, anyway.

MAIN BREAKDOWN

The function is continuos and can be differentiable at all points or not is an important factor.

Differenciable / Not Differentiable

Objective (generally LOSS) function

DIFFERENCIABLE

1st ORDER : Slope of objective

↳ GRADIENT : Derivative of a multivariate continuos obj. func. (vector)

↳ PARTIAL DERIVATIVE : Element of the gradient vector

2nd ORDER : Rate at which derivative of the function changes

↳ HESSIAN MATRIX : Multivariate 2nd derivative

⇒ There are many ML algoritms (almost all that I know) use 1st order derivative for optimization. AS :

- Gradient Descent ← - Stochastic Gradient Descent
  - Batch Gradient Descent
  - Mini-batch Gradient Descent
- Momentum
- Adagrad
- RMSProp
- Adam

Gradient Descent (core)

→ We have a cost function differentiable and has minimas.

→ Start with an initial value for model parameters, find the cost function, find the derivative (gradient) of the cost function.

→ coefficient = coefficient − (α . Δ)

where α = learning rate

→ We go downhill as we know the directions by the gradient and we move coefficients in that direction as much as it is allowed by learning rate.

(1) Batch Gradient Descent — SLOW —

→ As Gradient Descent is used for the entire dataset at each iteration → Batch Gradient

(2) Stochastic Gradient Descent — FAST —

→ At each step, it takes a random record, uses it to update coefficients and repeats this process.

→ Jumps a lot, overshooting → DECREASE LEARNING RATE

(3) Mini-Batch Gradient Descent — MIDDLE —

→ Takes a set (with smaller number than the train set) and take gradient for these data each step, update parameters and repeat this process.

→ Choose mini-batch size wisely (Generally 50-256)

NOTE: mini batch size = 1 → Stochastic Gradient Descent

>> Learning rate & Other parameter seclection is important! <<

* Gradient Descent is utilized in different forms in terms of selection of size in the batch.

⇒ HOWEVER, there are lacks in these methods, and several optimizers built on top of these.

Momentum – with SGD –

* That helps accelerate SGD in the relevant direction and damps ossicalions.

* Adds a fraction γ of the update vector of the past time step to the current vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

γ = momentum (often 0.9)

↳ With this if gradient changes direction when compared to last round, velocity is slowed

↳ If it does not; velocity is increased

FOR RELEVANT PARAMETERS.

* Nesterov Accelerated Gradient *

By replacing θ with $\theta - \gamma v_{t-1}$:

$$v_t = \gamma v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

→ We give an idea of the future update, make the calculation with looking ahead to approximate future positions of our parameters instead of currents.

→ SMARTER dropping ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Adagrad

→ It updates the learning rate for each parameter at each step.

* Low learning rates for → frequently occuring features
* High learning rates for → infrequent features

→ Well suited for sparse data.

* $g_{t,i} = \nabla_\theta J(\theta_{t,i})$ at time t, for parameter i

* $\theta_{t+1,i} = \theta_{t,i} - \eta \, g_{t,i}$ } SGD update

↳ $$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \odot g_{t,i}$$

where $G_{t,ii}$ ~ past parameter gradients (sum of squares)

ϵ ~ smoothing term to avoid division by zero ($\sim 1e-8$)

→ Accumulation of squared gradients in the denominator keeps growing and it leads to diminishing learning rate, which, in some cases, might result in model getting stuck.

↙

Adadelta

→ Built on Adagrad but with an attitude of reducing aggressive, monotonically decreasing learning rate.

* Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

→ More Efficient: Don't store w previous squared gradients. Store it decaying average of all past squared gradients.

• $E[g^2]_t = \gamma \cdot E[g^2]_{t-1} + (1-\gamma) g_t^2$ | $\gamma \sim$ momentumish

Then :

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t+\epsilon}} \odot g_t = \frac{-\eta}{\sqrt{E[g^2]_t+\epsilon}} \cdot g_t = \frac{-\eta}{RMS[g]_t} \cdot g_t$$

→ $E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta^2_t$

↳ $RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t+\epsilon}$

* which we do not know ! Need to be estimated

$$\Delta\theta_t = \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} \cdot g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

→ Adadelta , we do not need to set a default learning rate as it has been eliminated from the update rule.

RMSprop

→ Again , works on Adagrad's diminishing learning rate.

→ RMSprop ~ IDENTICAL ~ to first update vector of ADADECTA !

$$E[g^2]_t = 0.9\, E[g^2]_{t-1} + 0.1\, g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t+\epsilon}} \cdot g_t$$

→ $\gamma = 0.9$ by Hinton (developed RMSprop) , while a good default value for learning rate ~ 0.001 .

Adam

→ Adaptive learning rates for each parameter as well.

→ exponentially decaying avg of past squared gradients

⊕ exponentially decaying avg. past gradients m_t, similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \sim \text{mean (gradients)}$$

$$V_t = \beta_2 V_{t-1} + (1-\beta_2) \cdot g_t^2 \quad \sim \text{variance (gradients)}$$

⇒ Initialized as 0 vectors. Biased towards zero.

TO ATTACK :

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad , \quad \hat{V}_t = \frac{V_t}{1-\beta_2^t}$$

$$\Rightarrow \boxed{\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{V}_t} + \epsilon} \cdot \hat{m}_t}$$

$\beta_1 = 0.9$
$\beta_2 = 0.999$
$\epsilon = 10^{-8}$
} Proposed default values

Adamax

→ V_t factor was for ℓ_2 - norm of past gradients BUT it can be EXTENDED to ℓ_p form.

$$u_t = \beta_2^{\infty} v_{t-1} + (1-\beta_2^{\infty}) |g_t|^{\infty} = \max(\beta_2 \cdot v_{t-1}, |g_t|)$$

$$\boxed{\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \cdot \hat{m}_t}$$

$\eta = 0.002$
$\beta_1 = 0.9$
$\beta_2 = 0.999$
} Proposed default values

N-Adam

* Nesterov - Accelerated Adaptive moment Estimation
* Combines Adam & NAG.

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - m_t = \theta_t - (\gamma m_{t-1} + \eta g_t)$$

Also: $g_t = \nabla_{\theta_t} J(\theta_t - \gamma m_{t-1})$ From NAG.

↳ But don't apply twice! So use $J(\theta_t)$

Instead

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

↳ when replaced with its values.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_t + \frac{(1-\beta_1) g_t}{1-\beta_1^t} \right)$$

AMSGrad

→ Exponential averaging prevent learning rates to become infinitesimally small.

→ BUT, in settings where Adam converges to a suboptimal solution, some minibatches provide large & informative gradients, but they are RARE, exponential avg. diminishes their influence

↳ POOR CONVERGENCE.

- $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$, $\hat{v}_t \Rightarrow$ NO!
- $\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot m_t$$

SECOND ORDER OPTIMIZERS

→ Univerite Obj. Functions

- Newton's method
- Secant method

→ Multivariate Obj. Functions

- Quasi - Newton Method

...

NON-DIFFERENTIAL OPTIMIZERS

→ Direct Algoritms

- Cyclic Coordinate
- Powell's method
- Hooke-Jeeves method

→ Stochastic Algoritms

- Simulated Annealing
- Evolution Strategy
- Cross-entropy method

→ Population Algoritms

- Genetic Algorithm
- Differential Evolution
- Particle Swarm Optimization

MORE ADVANCED TOPICS

CONCEPT OF ENTROPY

* In Information Theory,

→ Entropy of a RV. is the average level of "info", "surprise" in its possible outcomes.

* For X with outcomes x_i with prob. p_i

$$H(x) = -\sum_{i=1}^{n} P(x_i) \cdot \log(P(x_i))$$ ENTROPY

- log base 2 bits
  base e nats
  base 10 dits ...

$$G(x) = 1 - \sum_{i=1}^{n} p^2(x_i)$$ GINI

- High Entropy: More uncertainty
- Low Entropy: more predictability

Information Gain

* Concept of a decrease in entropy after splitting the data on a feature.

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

- I = either entropy or gini
- D_j = dataset on the j^{th} split
- D_p = dataset on the parent node
- N = # of data in parent
- N_j = # of data in j^{th} child

⇒ Decision Trees split by looking at IG!

VALIDATION

* We need to validate our trained model with a hold-out dataset to test how it performs outside-the box.

Simple Train-Test Split

* Split the data in two parts
  - ↳ use 1 to train
  - ↳ use other to test
* You can stratify!
* PROBLEMATIC for MULTICLASS / MULTILABEL data
  - ↳ StratifiedShuffleSplit

CROSS - VALIDATION

(1) To compare different models or different hyperparameters as a part of
- different model error stats & their deviations
- some model different hyperparameters error stats & deviations

↳ Then, DECIDE which model is more powerful AND Stabile.

(2) Small set of data and you cannot split a test data completely seperte → LOSS OF INFORMATION

↳ Do CROSS-VALIDATION :

* If the error is satisfying and
* Error does not fluctuate a lot !

↳ Train a model on the whole dataset (FINAL MODEL)

Leave-One-Out cross validation

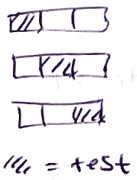
Training Data = n-1
Test Data = 1
} n times repeat

↳ Less bias
↳ computationally expensive

k-fold cross Validation

This time, data is folded into k parts, each k time:

- Train: k-1 parts
- Test: k^{th} part

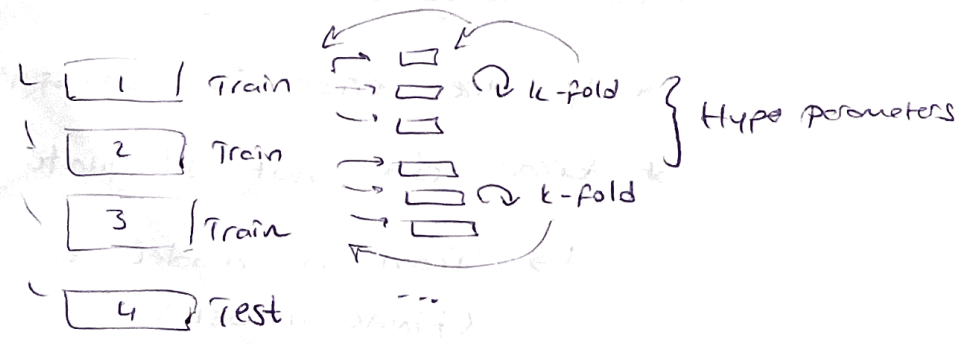


} repeat k times

⇒ When k=n → LOOCV!

* You can stratify to deal with imbalanced datasets
* You can also shuffle dataset each loop.
* WHAT IF:
  * You have a small dataset where you have to build a model and tune your hyperparameters?
    - NESTED k-fold CV
      ⇒ k-fold CV for hyperparameter optimization is NESTED INSIDE the k-fold CV for model selection.

1st



Then you find best hyperparameter sets, REFIT them to all train 1st layer data
↳ Test those models → repeat
↳ REACH a optimized set with model stats.

→ But it might be costly in terms of the INCREASE in the NUMBER OF MODEL EVALUATIONS performed.

↳ k · n - k

PYTHON IMPLEMENTATION

* sklearn.model_selection.

↳ . train_test_split (X, y, test_size = ?,
Shuffle = False / True,
random_state = ?,
Stratify = array / None) = $X_{tr}, y_{tr}, X_{test}, y_{test}$

↳ . KFold (n_splits = ?,
Shuffle = True / False,
random_state = ?) } Object

→ . split (X, y = None, groups = None) = train, test (index)

↳ . StratifiedKFold (n_splits = ?,
Shuffle = True / False,
random_state = ?) } Object

→ . split (X, y, groups = None) = train, test (index)

↳ . cross_val_score (estimator,
X, y = None,
groups = None,
scoring = 'str' or score callable (est., X, y),
cv = None (5),
n_jobs = None / -1,
verbose = 0,
fit_params = None,
error_score = 'raise', np.nan) → Scores

↳ RepeatedKFold (n_splits, n_repeats, random_state)
↳ LeaveOneOut()
↳ GroupedKFold(---)
...

ALSO

* Sklearn. cross-validation.

↳ Looks like same functions are defined here as well.

⇒ Write code for further xd

BIAS-VARIANCE TRADEOFF

BIAS: The error that is introduced by modelling a real-life problem.

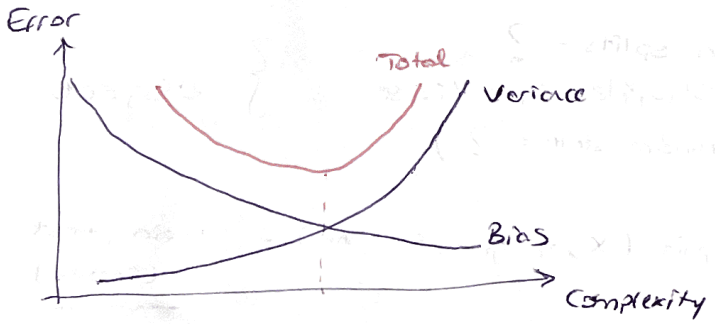
→ Our prediction vs truth difference

→ High bias = Simple model (over!!!)

VARIANCE: Variability of model prediction for given data.

→ High variance = Complex model that overfits

$$Err(x) = Bias^2 + Variance + \varepsilon$$ → minimize!



* Low bias good fit but as it goes too low as the model fits more and more, variance increases.

↳ Find the optimal by testing your model with hold-out-sets and comparing metrics.

↳ Fix your model / hyperparameters if NEEDED!

⇒ FIND THE BEST BALANCE!

⇒ NO OVERFIT / NO UNDERFIT!

SUPERVISED MODELS

* Data is labeled and labels are utilized during the model parameter optimization process.

  ↳ CLASSIFICATION : Target is finite and unordered set.

  ↳ REGRESSION : Target is continuous and quantitative.

CLASSIFICATION

① Binary classification
② Multi-class Classification
③ Multi-Label Classification

} x Also combination of these

LOGISTIC REGRESSION

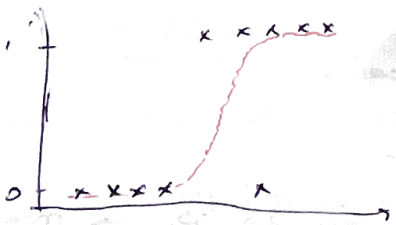
→ It does work as binary classification model, but also gives probabilities to be used as ranking.

→ Hypothesis $z = wx + B$

$h(x) = \text{sigmoid}(z)$ where $\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$

$= P(Y=1 \mid x; \text{theta})$

→ Data is fit into linear regression model, which then be acted upon by a logistic function predicting the target categorical dependent variable



→ S-shaped curve into data that gives (0,1) probability function.

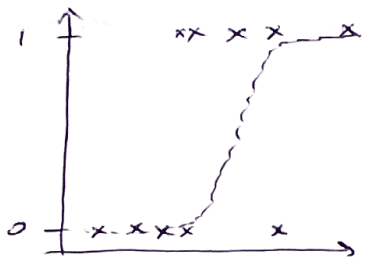
→ Uses maximum likelihood and calculates it at each iteration until maximizing it !

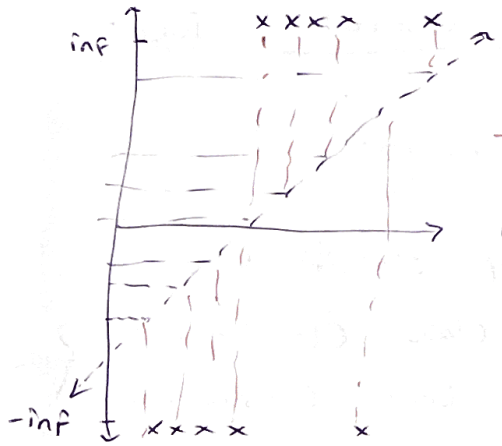
→ Logit $= \log\left(\frac{p}{1-p}\right)$ function reverses it into linear line fit.

→ Intercepts are in linear function. | (after logit)
Coefficients 0

→ When categorical, log(odds) used again also difference in log(odds)s.

* COEFFs are in terms of log(odds) | 0





→ residuals farar away
↳ CANNOT USE LEAST SQUARES

→ Instead we project these points onto the line and find log(odds) correspondents.

→ Transform candidate log(odds) into probabilities by sigmoid function $p = \frac{e^{\log(odds)}}{1 + e^{\log(odds)}}$

→ Likelihoods are predicted probabilities. (p) on the line.

⇒ Take the products of likelihoods for each class:

- Positive class $\pi(p)$
- Negative class $\pi(1-p)$

} All multiplied

BUT: Log-Likelihoods

↳ log (Pf) .....

→ Rotate the line → go to Ps → max log-likelihood

* NO OUTLIARS
* NO MULTICOLLINEARITY IN PREDICTORS

→ Uses log-loss function so that gradient descent can be used to find a minima in convex space.

PYTON

* sklearn.linear-model.LogisticRegression

```
( penalty = 'l1', 'l2', 'elasticnet', 'None',
  dual = False,
  C = 1.0,
  class_weight = None,
  random_state = ?,
  solver = newton-cg / lbfgs / liblinear / sag / saga
  multi_class = 'auto' )
```

- coef_
- intercept_
- n_features_in_
- feature_names_in_
- n_iter_

- fit(x, y)
- predict(x)
- predict_proba(x)

Naive Bayes Classifier

Based on Bayes' Theorem, with the assumption of independence between every pair of features.

→ Extremely fast.
→ Can work with small set of data.
→ Document classification and spam filtering

* Beginning of this NOTEBOOK!

GAUSSIAN

A priori probability of class A $P(A)$ is used. Also $B \to P(B)$

$P(A) \times L(X_1 = x_1 \mid A) \times L(X_2 = x_2 \mid A) \ldots = P(A \mid X_1 = x_1 \ldots)$

$P(B) \times L(X_1 = x_1 \mid B) \times L(X_2 = x_2 \mid B) \ldots = P(B \mid x_1 = x_1, \ldots)$

where likelihoods are pdf values of Normal Gaussian of features.

↳ MAY USE LOG() function log() + log(1 ...

* WHICH SCORE higher → classified

PYTHON

* sklearn.naive_bayes.GaussianNB

()

- fit(x, y)
- predict(x)

↳ can be easily writable by self.

K-Nearest Neighbors

→ Decide a number k which will be the nearest neighbors number to ask their vote.
The MAJORITY class voted is the data's class.

* Simple, used for nonlinear data
* Gives high accuracy but there are better models
* It stores ALL training data and calculates distances then orders and decides.

* Decide k with cross validation

Python

* sklearn.neighbors.KNeighborsClassifier

(algorithm = 'auto',
leaf_size = 30,
metric = 'minkowski',
metric_params = None,
n_neighbors = 5,
p = 2,
weights = 'uniform')

- fit(x, y)
- predict(x)

DECISION TREE

→ Creates rule patterns.

→ Top-down, greedy recursive binary splitting.

* It uses previous concepts of gini and entropy to calculate impurities, ergo information gain on splits.

  ↳ Maximizing IG is the selection method for feature and cut points.

→ No need to normalize

→ can be used as preprocessing tool for missing value prediction.

→ No assumptions about distribution

→ Might overfit on noisy data OR high dimensional scarce data

→ Biased with imbalance dataset

PRUNING

→ We can grow the tree as the improvement criteion exceeds a threshold.

  * BUT SHORTSIGHTED: an unimportant split can be followed by an important one.

→ First, build the model.

→ Then prune with COST COMPLEXITY PRUNING

$$\sum_{m=1}^{|T|} \sum_{i:x_i} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

→ Eliminate weakest link

PYTHON

* sklearn.tree.DecisionTreeClassifier(
  criterion = 'gini' / 'entropy',
  splitter = 'best'
  max_depth = None,
  min_samples_split = 2 (number OR fraction))

- feature_importances_
- max_features_
- n_features_
- feature_names_in_
...

- fit(x, y)
- predict(x)

* sklearn.tree.DecisionTreeClassifier()
  - cost_complexity_pruning_path(x, y)
    - ccp_alphas
    - impurities

    } plot

↳ Build a classifier set with ccp_alpha = ccp_alpha;

↳ Run and calculate accuracy for training and test sets. OR CV!

Regressor Tree

* Instead of majority count, takes the mean of the leaf to predict.
  * DecisionTreeRegressor(
    criterion = squared_error / mse / absolute_error / mae,
    ...)

But ONE TREE IS NOT THE BEST!

BAGGING ~ Bootstrap Aggregating ~

* Bootstrapping = Resampling with replacement

→ Here, same as the overall sample size, create B bootstrapped samples.

→ Fit each one a tree.

→ Aggregate each tree outcome by majority vote among trees.

* Reduces variance...

→ Each weak learner is constructed parallelly, not on top of each other.

→ Standard = Bootstrapped Sample Size = Sample size

Out-of-Bag (OOB) Error

→ Since sampling is done by replacement, some records occur more than once, some are never in the bag.

~ Appx. 1/3 is Out-of-Bag.

→ Train the model on the bag and predict the records that are OOB and measure the metric of interest.

↳ The average of all metrics for each bag is the model metric. We also have deviation

≈ kinda CROSS-VALIDATION

Python

* sklearn.ensemble.BaggingClassifier(

```
base_estimator = DecisionTreeClassifier,
n_estimator    = # bases,
max_samples    = bag size (can be %),
max_features   = featureset size (can be %),
bootstrap      = True/False,
bootstrap_features = True/False,
oob_score      = True/False,
random_state   = ? ).
```

- base_estimator_
- n_features_
- n_features_in
- feature_names_in
- estimators_
- estimators_samples_
- estimators_features
- classes_
- oob_scores_
- oob_decision_function_

- fit(x, y)
- predict(x)
- predict_proba(x)

⇒ You can create different base models etc...

RANDOM FOREST

→ Built on the idea of bagging.

→ Takes many trees from bootstrapped samples but this time, at each sample, a random sample of m predictor is chosen as candidates from full set of p.

convertionally $m \sim \sqrt{p}$

* If the are strong predictors among some moderate ones (in real life always :)), bagging will select strongs as the first split at each learner.

↳ use a random subset of features.

↳ DECORRELATE TREES

PYTHON

```
Sklearn.ensemble.RandomForestClassifier(
        n_estimators = 100,
        criterion = gini / entropy,
        max_depth = None,
        min_samples_split = 2,
        min_samples_leaf = 1,
        max_features = auto,
        max_leaf_nodes = None,
        bootstrap = True,
        oob_score = False,
        class_weight = 'balanced'.... )
        ccp_alpha = 0.0
        max_samples = None
```

In addition to bagging:

. feature importances

. fit(X, Y)

. predict(x)

. predict_proba(x)

ISOLATION FOREST

* Built on the basis of decision trees, mostly for outlier detection.

→ If you can seperate a sample in the early cuts then it means it is more away from other values, potential OUTLIERS! If a sample is in a deep leaf of a tree, it means it was hard to distinguish it from others (regular values)

$$\text{Anomaly score} = s(x,n) = 2^{-\frac{E(h(x))}{c(n)}}$$

$h(x)$: path length of obs. x

$c(n)$ = avg path length of unsuccessful search in a Binary Search Tree

n = number of external nodes

→ After fitting the algorithm, it calculates anomaly scores for datapoints and with thresholds it plots them.

Python

* sklearn.ensemble.IsolationForest(
  - n_estimators = 100,
  - max_samples = auto,
  - contamination = auto,
  - max_features = 1.0,
  - bootstrap = False,
  - random_state = None)

previous tree based returns + offset -

- fit(X)
- fit_predict(X) → +1/-1
- predict(X) → +1/-1
- decision_function(X) → anomaly scores

⇒ It is somehow unsupervised!

↳ No labels as outlier/no outlier...

EXTENDED ISOLATION FOREST

→ Again UNSUPERVISED, same mechanism but extended.

→ Isolation Forest has only vertical / horizontal cuts. Therefore if there are multi-modes in the data then may mislead.

↳ NEVER USE SIMPLE ISOLATION unless the problem is too simple and obvious.

↳ ALTHOUGH, if you are doing to a simple feature each time (1-dimension) then OKAY.

PROCEDURE

* Random slope for the branch cut
* Random intercept chosen from the range of available values from the training data

SELECTED.

* Makes tailored hyperplanes, better results

Python

eif.iForest (x,
ntrees,
sample-size,
ExtensionLevel)

By ExtensionLevel we can create Extended ISO.

BOOSTING

* Trees are grown seqvertially, each tree grown using information from previously grown trees.

→ DOES NOT HAVE TO BE SET OF TREES. H can take other models as well.

PROCEDURE

1 → Start with initial equal weights to samples
2 → Take randomly one, fit the tree/model, store predictions & RESIDUALS
3 → Give more weight to samples with higher residuals
4 → Repeat from 2.

$$\text{Amount of say} = \frac{1}{2}\log\left(\frac{1 - \text{Total Error}}{\text{Total Error}}\right) \rightarrow \text{sample weights}$$

ADAPTIVE BOOSTING

* In ADABOOST, trees are only stumps contrary to RF where tree lengths mish be long and variant among weak classifiers.

o STUMPS = 1 noded binary splits

PROCEDURE (numbers example)

1. Divide data to length of data to give initial sample weights.
2. Make the first stump by finding the variable that does the best job. For each variable - split - find gini/entropy & SELECT the best one.
3. Sum of the weights classified as wrong → TOTAL ERROR! = $1/8$
4. Calculate Amount of Say = $\frac{1}{2}\log\left(\frac{1 - 1/8}{1/8}\right) = \frac{1}{2}\log(7) = 0.97$
5. New sample weight = sample weight $\times e^{\text{amount of say}}$ (INCORRECT) $= \frac{1}{8} \cdot e^{0.97} = 0.33$
6. New sample weight = sample weight $\times e^{-\text{amount of say}}$ (CORRECT) $= \frac{1}{8} \cdot e^{-0.97} = 0.05$
7. Normalize sample weights as summing up to 1.
8. Pick a random number and use sample weights as cdf for sample selection

9. Bootstrap sample as the same size with original with this random selection process.

   * Notice weight-increased samples of incorrectly predicted will appear more in bootstrapped samples.

10. Repeat this process until n_estimators are reached.

⇒ Amount of says calculated are also used as vote weights of the stumps in the final decision.

⇒ class P votes $\sum$ amount of says
class N votes $\sum$ amount of says
} compare ⇒ CLASS

PYTHON

* sklearn.ensemble.AdaBoostClassifier(

base_estimator = DecisionTreeClassifier(max_depth=1),
n_estimators = 50,
learning_rate = 1.0,
algorithm = SAMME.R / SAMME,
random_state = None)

Addition:
. estimator_weights_
. estimator_errors_
. feature_importances_
...

- .fit(x, y)
- predict(x)
- predict_proba(x)

GRADIENT BOOSTING

→ The idea is the same. But this time, weights are not the manipulation tool.

→ A Loss function is defined as it will give us the residuals for our predictions.

→ Then, the next weak learner tries to fit the model to those residuals.

PROCEDURE

1. Define a differentiable LOSS FUNCTION. that is compatable with base estimator.

2. Initialize your model $F(x)^0 = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$

3. For m=1 to M:

   - Compute pseudo-residuals $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]$

   - Fit a base learner $h_m(x)$ to residuals using the train set $\{(x_i, r_{im})\}_{i=1}^{n}$

   - Compute multiplier $(\gamma_m) = \underset{\gamma}{\text{argmin}} \sum L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$

   - Update the model $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$

4. Repeat 3 until n-estimators reached.

⇒ By this, we adjust our rate in residual-fitted model whilst minimizing the loss function;

* AND we add it to the previous model, basically ADJUSTING the previous model more accordingly to its MISTAKES.

⇒ Does not have to be stumps! But different trees combined.

⇒ Named as Gradient since it uses minimizing loss function by gradient descent.

PYTHON

* Sklearn.ensemble.GradientBoostingClassifier(

  loss = deviance / exponential,

  learning_rate = 0.1,

  n_estimators = 100,

  subsample = 1.0 (smaller than 1.0 → SGD)

  criterion = friedman_mse / squared_error, mse, mae,

  min_samples_split = 2,

  min_samples_leaf = 1,

min_weight_fraction_leaf = 0.0,
max_depth = 3,
min_impurity_decrease = 0.0,
random_state = None,
max_features = None / auto / sqrt / log2,
verbose = 0 (>1 prints each tree),
max_leaf_nodes = None,
validation_fraction = 0.1,
n_iter_no_change = None, // early stop //
tol = 1e-4,
ccp_alpha = 0.0)

- oob_improvement_
- train_score_
- loss_
- ... other ones.

XGBoost

→ Gradient Boosting is SLOW since it is SEQUENTIAL.

→ XGBoost is the same but further developed to solve performance / efficiency issues.

* parallelization of tree construction using all CPUs
* Distributed computing using cluster of machines
* Out of Core computing that data not fitting into memory
* Cache Optimization to make the best use of hardware

* xgboost.XGBClassifier(
  n_estimators
  use_label_encoder
  max_depth
  learning_rate
  verbosity,
  objective = 'binary:logistic',
  booster
  tree_method = 'auto',
  gamma = minimum loss reduction required,
)
  + lambda = L2 reg. (=1)
  alpha = L1 reg. (=0)

  >>>

  .fit(x, y)
  .predict(x)
  .predict_proba(x)

Objective

- binary:logistic → lg regression for binary prediction → prob.
- multi:softmax → multiclass classification using softmax → class
  - ↳ num_class
- multi:softprob → same as softmax but returns probs.

Errors (eval-metric)

- rmse
- mae
- logloss (neg. log-likelihood)
- error
- merror (multiclass)
- mlogloss (multiclass)
- auc

LIGHT GBM

→ Microsoft released in 2017

→ Others split depth-wise whereas Light GBM splits leaf-wise

* Finding the best split for each leaf is really hard. Naively done → O(n.p) complexity.

TECHNOLOGIES

* Gradient Based One Side Sampling
  - Sort the instances acc. to |gradients| in descending order
  - Select the top a×100% instances
  - Randomly sample b×100% from the rest of the data
    - ⇒ Putting more weight on undertrained without disrupting data distribution
* Exclusive Feature Bundling
  - Construct measure of conflict between features (based on overlapping none zero values)
  - Sort the features by count of non zero instances in descending
  - Loop over the ordered list of features and assign the feature to an existing bundle (conflict < threshold)
    new bundle (conflict > threshold)

→ This helps in dimension reduction (somehow) and speed up the process.

→ Some features are never non-zero together.

Python

```
lightgbm.LGBMClassifier(
        boosting_type = 'gbdt' ....,
        num_leaves    = 31
        max_dept      = -1
        learning_rate = 0.1
        n_estimators  = 100
        subsample_for_bin = 200,000,
        objective     = None /default/,
        reg_alpha     = L1,
        reg_lambda    = L2,
          ...                    )
```

- objective –
- evals_result –
- booster –
- best_iteration –
- best_score –

.fit(x, y)
.predict(x)
.predict_proba(x)

HYPERPARAMER TUNING

RULES

* Know which model you are working with and what that might suffer from.
  ↳ Depth, criterion, n-features etc.

* Know your data and its size, have good propositions for hyperparameters

* Create a dictionary (set) of variables + their proposed values / by variables = hyperparameters /

↳ Run a CV / OOB on data whilst testing different sets of hyperparameters.

↳ KNOW your SEARCH Algorithms

GRIDSEARCH

→ You can construct a dict with list of proposed values as values and hyperparameters as the keys
  * Then construct your grid search with CV by python with loops etc.

→ Already constructed functions : GRIDSEARCH(CV)

* sklearn.model-selection. GridSearchCV(

```
estimator =      ,
param-grid = dict ,
scoring   = 'compatible score name' ,
cv        = 5 ,
verbose   = 1 ,
n-jobs    = -1 )
```

- fit (x, y)
- get-params ()
- inverse-transform (x)
- predict (x)
- predict_proba (x)

} best

- cv_results_
- best_estimator_
- best_score_
- best_params_
- best_index_
- scorer_
- n_splits_
- multimetric_

RANDOM SEARCH

* Create a grid and select random combinations among these:

  * Sklearn.model_selection.RandomizedSearchCV(

    estimator = ,
    param_distributions : dict / list of dicts ,
    n_iter = 10 ,
    scoring = None ,
    n_jobs = None ,
    refit = True ,
    cv = None (s) ,
    verbose = int ,
    random_state = None)

    } Some Grid.

  → Can take whole distributions as parameter value ranges in the dict.

BAYESIAN OPTIMIZATION

* Hyperopt.fmin()

  → It uses probability to find the minimum of a function (loss)

    (objective_function = loss
      domain_space = range of inputs ,
      optimization_algorithm =)

  → Better than others.

SUPPORT VECTOR MACHINES

→ Can be applied to a variety of settings inc. binary classification, multi-nominal classification, regression.

→ Separating hyperplanes

PROCEDURE

1. Fit a n-1 hyperplane to an n-dimension space
2. Measure perpendicular distance to the hyperplane of each data
3. Take the smallest distance → margin
4. Find a hyperplane that maximizes the margin.

↳ works if seperatable classes

What if not-seperating classes

SOFT MARGINS

* It allows some points being on the ~~right~~ wrong side of the hyperplane.

* Penalty term C:
  - Larger, more sophisticated models.
  - If kernel not chaged, then a wider margin comes with larger C (penalty for misclassification) → ↑ support vectors

$$\max \quad M \quad \text{s.t.} \quad \sum_{1}^{p} \beta_j^2 = 1$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}) \geq M(1 - \epsilon_i)$$

$$\epsilon_i \geq 0, \quad \sum_{1}^{n} \epsilon_i \leq C,$$

KERNEL TRICK

* for finding non-linear desision boundries.

* change kernel →
  - linear
  - poly
  - rbf
  - sigmoid
  - precomputed

RBF

$$\phi(x, center) = \exp(-\gamma \|x - center\|^2)$$

→ Transformer by measuring the distance between all other dots to a specific/dots-centers.

γ = rate of influence of new features
↳ more, more wissly

KERNEL: Data is projected/trasformed into more ordered, standardized forms. During this we do not store new coordinates. Except, we store:

- Distances:

$$\langle x_i, x_{i'} \rangle = \sum_{1}^{p} x_{ij} x_{i'j} \quad \text{(inner product)}$$

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle$$

↳ It turns out many α_i is 0.
↳ S = support set of i, $\alpha_i > 0$.

Linear → $x_i \cdot x_j$

Polinomicl → $(1 + x_i \cdot x_j)^d$

Radial → $\exp(-\gamma \|x_i - x_j\|^2)$

Sigmoid → $\tanh(\alpha x^T y + c)$

PYTHON

* sklearn.svm.SVC (

c = 1.0,
kernel = 'rbf',
degree = if poly,
gamma = 'scale',
coef0 = if poly/sigmoid,
shrinking = True,
probability = false,
tol = 1e-3,
...,
decision_function_shape = ovr/ovo)

- support_
- support_vectors_
- n_support_
- probA_ / probB_
- shape_fit_
- ---
- fit(X, Y)
- predict(x)

UNSUPERVISED MACHINE LEARNING

* No labels, finding pattens in the data.

→ Paronetric : Gaussian Mixture Models / Exp.-Max. Algos

→ Non-paronetric : As we know it, dist.- free

* CLUSTERING

└ Exclusive (K-Means), Overlapping (Fuzzy-K-Means), Hierarchical (Hierarchical), Probabilistic (Mixture of Gaussian)

K-MEANS CLUSTERING

→ This algorithm basically finds k cluster centroids and assign points to the cluster with the minimum distance.

• PROCEDURE • Greedy

1. Randomly select k centroids
2. Associate each point to the clasest centroid
3. Minimite $J = \sum_{j=1}^{k} \sum_{i=1}^{x} \| x_i^{(j)} - c_j \|^2$

distance btw a data point x_i and the cluster centre c_j

4. Re-calculate cluster centers:

$$v_i = \left(\frac{1}{c_i}\right) \cdot \sum_{j=1}^{c_i} x_i$$

where c_i = # of points in cluster i

5. Repeat until no centroid is moved.

- Initialitation is not specified.
- Set of samples closest to v_i is empty, so it cannot be updated. → Handle at implementation

→ use Elbow-method to deide K.

→ Scaling might be useful in improving K-means.

→ You cannot have CATEGORICAL vaicbles

They should be NUMERIC!

$\sim O(\#iters * \#clusters * \#instances * \#dimensions)$

! SCALE !

PYTHON

- scipy.cluster.vq → kmeans(

obs = data,
k-or-guess = int,
iter = 20,
thresh = 1e-05,
check_finite = True,
seed = None)

= centroids, _

* df['labels'], _ = vq(data, centroids)

→ whiten(data) → variation reduction

- sklearn.cluster.KMeans(

n_clusters = 8,
n_init = 10,
max_iter = 300,
...
)

- cluster_centers_
- labels_
- inertia_
- n_iter
- n_features_in
- feature_names_in

* MiniBatchKMeans()

- fit(X)
- fit_predict(X)
- predict(X)
- score(X)

HIERARCHICAL CLUSTERING

This is a method that starts with considering each data point as a cluster and then goes until there is only one cluster.

PROCEDURE

1. N clusters start.
2. Find the most closest clusters & merge them into one. → 1 Less clusters ←
3. Compute distances and REPEAT 2 until done.

DENDROGRAM

→ Visualization for hierarchical clustering.

→ Best to decide on number of clusters

* HORIZONTAL LINE ON DENDROGRAM.

PYTHON

- scipy.cluster.hierarchy

↳ linkage(

y = data,

method = 'single',

metric = 'euclidean',

optimal_ordering = False) = Z

↳ dendrogram(Z) → image

.plt.show()

↳ fclusters(Z,

criterion = 'maxclust'

n~~an~~clusters=)

DIMENSIONALITY REDUCTION

→ When we have too many features (even small number can be too many), we may have to reduce dimension for avoiding information loss (clustering) or overfitting.

FEATURE ELIMINATION

→ Variance Inflation Factor : Elimination method.

→ Univoiate Feature Selection : One-by-one ~ SFA

→ Recursive Feature Elimination : fits a model, eliminate k weakest features.

→ Model Based Feature Selection : feature importances etc.

* Loosing some information since features are directly dropped without any INFO EXTRACTION!

PRINCIPAL COMPONENT ANALYSIS

→ Feature Extraction & Dimensionality Reduction Technique

→ Might be used as a tool that takes features and get them into 2/3D visualization.

PROCEDURE

① Calculate a matrix that summerizes how our variables are all relate to one another. with direction and magnitude

② Transform our orijinal data to alisn with these important directions.

↳ Do it for all features.

↳ Orthogonal directions.

→ X matrix, each column is standardized to have 0 mean. (-mean for sure, but standardizes depends on your selection)

→ Calculate covariance matrix = $Z^T Z$

→ Calculate the eigenvectors by $Z^T Z \rightarrow PDP^{-1} \rightarrow (P)$

→ Take eigenvalues $\lambda_1, \lambda_2, \dots \lambda_p$, sort them in descending → (P^*) (sorted)

→ Calculate $Z^* = ZP^*$ is a rotated / standardized version of X but now each obs. is a combination of the original variables,

Weights = determined by eigenvector

↳ EACH COLUMN of Z^* INDEPENDENT!

↳ LINEAR INDEPENDENCE OF VARIABLES!

How MANY PCs?

→ Arbitrarily Select (~ visualization 2D, 3D)

→ Proportion of Variance Explained
- Threshold, add features until it is reached.

→ Scree Plot
↳ variance → elbow
↳ Cumulative → } again prop. of var. explained

* PCR (Principal Component Regression)
* Kernel PCA

PYTHON

* sklearn.decomposition.PCA(
  n_components = None,
  copy = True,
  whiten = False,
  svd_solver = 'auto',)

- components_
- explained_variance_
- explained_variance_ratio_
- singular_values_
- mean_
- n_components_
- n_features_
- n_samples_

- fit(x)
- fit_transform(x)
- transform(x)
- inverse_transform(x)

+ KernelPCA()
  SparcePCA()
  TruncatedSVD()
  IncrementalPCA()

t-SNE

* t-distributed Stochastic Neighbor Embedding

→ PCA may require data in a linear relationship. If we have non-linear relationship → t-SNE.

→ It calculates a similarity measure based on the distance between points instead of trying to maximize variance.

PROCEDURE

(1.) Measure similarities btw points in the high dimension.
For each data point (x_i) we center a Gaussian dist. over that point. Then we measure the density of all points (x_j) under that Gaussian.
└ Renormalize for all points.
└ Gives p_{ij} for all points ~ similarities

- use perplexity to adjust variance.

(2) Use Stunded t (cauchy) dist. with n-1 d.o.f. This gives us:
  - Second set of probs (Q_{ij}) in low dimensional space.

(3) We want Q_{ij} to reflect P_{ij} as best as possible.
  - Kullback-Liebler (KL) Diversence

* Use Gradient Descent to minimize KL cost function

- Cannot extend to new samples.
- Each run different results but relative position is same.

PYTHON

* sklearn.manifold.TSNE(
    n_components = 2,
    Perplexity = 30.0,
    early_exaggeration = 12.0,
    learning_rate = 200.0,
    n_iter = 1000,
    metric = 'euclidean',
    init = 'random', 'PCA',
    ...
)

  - embedding_
  - kl_divergence_
  - n_features_in_
  - feature_names_in
  - n_iter_

  - fit(x)
  - fit_transform(x)

! NO: .transform() ✗

* sklearn.metrics.silhouette_score