Cheatography

M5 Machine Learning Cheat Sheet

by anaischia2014 via cheatography.com/217080/cs/47591/

Supervised learning

Uses **labelled training data** with mapped features to known labels/targets to predict outcomes for new unseen data (the test set)

Classification: predicts categorical outcomes

Logistic regression: type of **parametric** classifier; passes a linear combination of inputs through a **logit (sigmoid)** function; **decision boundary** classes everything to left as 0 and right as 1; data is **not linearly separable** producing a non-zero error rate

Regression: predicts continuous outcomes

Cross-validation: for tuning hyperparameters and choosing between models, prevents overfitting or data-leakage by separating from test data

Scaling

Brings features into comparable ranges leading to faster and more stable model convergence i.e. distance-based algorithms

Normalisation: constrains values to a fixed range e.g. [0,1] or [-1,1]; MinMaxScaler() or Normalizer()

Standardisation: transforms the mean to 0 and variance/sd to 1 (z-scoring), data is unitless; StandardScaler()

Evaluation metrics (linear regression)

R^2: proportion of variance explained by model features; closer to 1 is better

MAE: average magnitude of errors and easily interpretable (same units as target), robust to outliers; smaller is better

MSE/RMSE: averaged squared difference between predicted and actual, sensitive to outliers; smaller is better

Evaluation metrics (classification)

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

False Positives are misdiagnoses, so precision gives *actual* TPs

False Negatives are missed diagnoses, so recall/sensitivity gives *identified* TPs

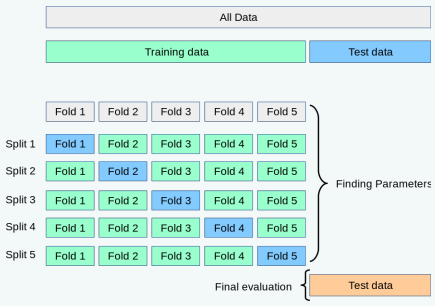
ROC-AUC: true positive rate vs false positive rate; closer to 1 is better

(**Specificity**: TN/(TN+FP)

Supervised Learning Pipeline

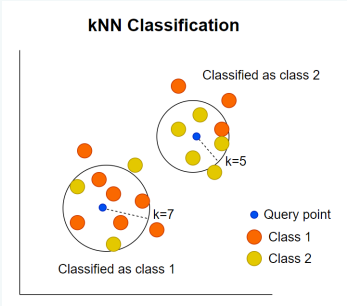
```
X = data.drop(columns='target')
y = data[['target']]
x_train, x_test, y_train, y_test =
train_test_split(X, y, test_size = 0.20..)
scaler = StandardScaler()
x_train=...scaler.fit_transform(x_train))
x_test=...scaler.transform(x_test))
model = LinearRegression()
model.fit(x_train, y_train)
y_pred = model.predict(x_test)
```

K-fold cross validation



Splits dataset into 'k' equal-sized folds, using k-1 folds for training and the remaining fold for validation, repeating k times to get an **average performance score**; **useful when data is limited** because every data point is used for both training and validation; **leave-one-out cross-validation** (LOOCV)

K-Nearest Neighbours (KNN)



Non-parametric classifier that looks at K points in training set nearest to test input x then computes average of these neighbours; **memory-based/instance-based learning**; works well given good distance metric (**Euclidean**) and sufficient training data; **poor performance under high dimensionality**; KNeighborsClassifier(n_neighbors=3).fit(X,y)

L1 vs L2 regularisation in regression

L1 (Lasso): sets some coefficients to 0 (feature selection); may jeopardise accuracy in small datasets

L2 (Ridge): shrinks coefficients and penalises higher weights

Parametric vs Nonparametric models

Parametric models: fixed set of parameters depending on number of features in e.g. regression, Naive Bayes or number of centroids e.g. k-means clustering; **faster performance but stronger assumptions**

Non-parametric models: makes no assumptions about dataset; number of parameters grow with amount of training data e.g. KNN, decision trees, random forest, kernel SVMs; **flexible but computationally expensive**

Unsupervised Learning

K-means clustering: uses euclidean distance (scale features!) and iteratively **minimises inertia** (within-cluster sum-of-squares)

k-cluster centroids chosen at random → each datapoint assigned to cluster with nearest centroid → each centroid updated by taking mean of all points assigned to that cluster

Elbow method: determines optimal number of clusters

```
kmeans = KMeans(n_clusters = 3, init = 'k-means++', max_iter = 300...)
y_kmeans = kmeans.fit_predict(feat_array)
```

Marginalisation

Sum of all probability values where X=x occurs with all possible values of Y

Information theory

If something is more likely -> less information; If unlikely -> more information

High entropy -> more uncertainty

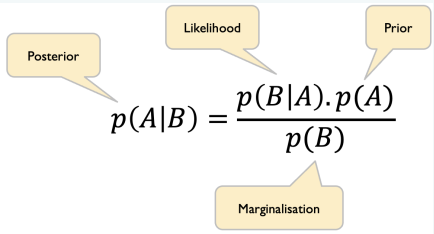
First-order Markov chain

Future is independent of the past given the present

Conditional probability

$$p(A|B) = \frac{p(A,B)}{p(B)} \text{ if } p(B) > 0$$

Bayes Rule



Base-rate fallacy: ignores prior probabilities of FPs, additionally use precision or confusion matrices

Naive Bayes

$$P(L \mid \text{features}) = \frac{P(\text{features} \mid L)P(L)}{P(\text{features})}$$

Assumes features are **independent**; requires **small amount of training data** to estimate parameters; aim is to predict **P(label | features)**; fast but bad estimator

Gaussian Naive Bayes classifier

```
GNB = GaussianNB(var_smoothing=0.5)
GNB.fit(x_train, y_train)
y_pred = GNB.predict(x_test)
y_pred_probs = GNB.predict_proba(x_test)
```

Compute **calibration curves** and **brier score** (lower is better), vary classification **decision threshold** (typically .5) and assess AUC, use GridSearch to vary var_smoothing parameter

Support Vector Machines (SVMs)

Supervised classifier that attempts to separate classes of data using a **hyperplane** wherein the 2 categories are **linearly separable**

Optimal hyperplane: maximises the **margin** between training points to minimise noise and hinge loss, preventing **overfitting**

Types of kernels: linear, poly, rbf, sigmoid - higher capacity and overfitting risk with more complex kernels

```
svc = SVC(kernel='linear'...)
svc.fit(X_train, y_train)
```

Compute accuracy and decision boundaries, use GridSearch to tune kernel and C hyperparameters (**large C = small margin**)

Pros: **high-dimensional spaces**; **memory-efficient** as uses support vectors; **versatile** with different kernels

Limitations: do not provide direct probability estimates; poor performance if no features > no samples

Decision Trees (DTs)

Nonparametric supervised learning for both classification and regression

Selects features iteratively based on a **criterion**: lowest *entropy*/highest information gain, *Gini impurity* i.e. how impure classes are within a dataset

node = feature, branch = choice, leaves = outcome

dt = DecisionTreeClassifier(...); dt.fit(X_train, y_train); Compute accuracy and decision boundaries; Use GridSearch to tune criterion and tree depth parameters

Limitations: prone to overfit; poor generalisability; high variance; slight changes in dataset can drastically change splits, complicating interpretation; unstable; errors at the top affect lower splits due to hierarchical nature; biased if dataset unbalanced

Random Forest

Ensemble model that consists of **multiple trees/base estimators**; overcomes limitations of DTs

Averaging: build several independent estimators and average predictions, **reducing variance or overfitting** in combined estimator

Pasting: random subsets of dataset are drawn as random subsets of samples

Bagging/bootstrapping: samples are drawn with replacement

Random Subspaces: random subsets of dataset are drawn as random subsets of features

Random Forest (cont)

Random Patches: base estimators are built on subsets of both samples and features

Boosting: base estimators built sequentially with the next/combined estimator trying to **minimise bias and underfitting**

XGBoost builds trees in parallel; *Gradient Boosting* minimises residuals sequentially (iterative)

rf = RandomForestClassifier(); rf.fit(X_train, y_train); Use pipeline for heterogeneous ensembles