Machine Learning Cheat Sheet

Introduction to Supervised Learning

- **K-Nearest Neighbors Classifier:** *test_size default None(0.25)*
  clf = KNeighborsClassifier(n_neighbors = 1) #def 5
  clf.predict(X_test)
  clf.score(X_test, y_test) #accuracy score
- **Cross-validation:**
  - **K-Fold: scores** - *kf = KFold(n_splits = k_folds, shuffle = True, random_state = 0).* ***#n_split = 5, shuffle = F***
    cross_val_score(logreg, iris.data, iris.target, cv = kf, n_jobs = -1)
  - **Stratified K-Fold:** *kf = StratifiedKFold(n_splits = k_folds, shuffle =* ***False****) #def n_splits = 5*
  - **Time-series split**
  - **Leave-one-out: loo** - *LeaveOneOut()*
  - **Shuffle and split** - *shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10).*
- **Grid Search cross validation:** identify best model parameterization, parameter_grid. cv is "None" for 5.
  grid_search = GridSearchCV(KNeighborsClassifier(), param_grid,cv = 5, return_train_score=True)
  *grid_search.cv_results_ / grid_search.score(X_test, y_test) / grid_search.best_params_ / grid_search.best_score / grid_search.best_estimator_*

Linear Model for Regression: Weighted sum of input features + bias
OLS = linear_model.LinearRegression() / OLS.intercept_ / OLS.coef_ / OLS.predict(X_test) / OLS.score(X_test, y_test)

- Fitting least squares: minimize the cost function RSS(residual sum of squares).
- Measure of fit: R2 0->1. No variance has been explained, 1 mean perfect fit to the data
- Regression performance metrics: R2(Coeff of determination), MSE / **RMSE**, MAD, MAPE
- **OLS issues:** Prediction Accuracy and Model Interpretability: Sensitive to outliers / no solution if # features(p) > #data points(n), not for high dimensional.
  Works well when linear. When n is almost equal to p, can overfit. P>n, variability increases dramatically.
- **OLS improvement:** Shrinkage / Feature Selection / Dimension Reduction
- Regularization is added to the cost function only during the training. Bias is not regularized, only weights.
  - **Tuning parameter**: alpha>=0, if very large, then more beta will tend to 0 – flat line through mean; if 0 then it is OLS. α controls the amount of shrinkage: if higher, greater the amount of shrinkage and thus the simpler the model.
  - When p>n, can use ridge
    *ridge = Ridge(alpha = 1.0).fit(X_train, y_train) / ridge.score(…) / ridge.coef_*
- **Ridge Regression Issues:** consider all predictors / coeff may changes substantially based on scale
- **Lasso** (Least absolute shrinkage and selection operator): Tuning parameter – alpha
  - Variable selection: with very large alpha, coeff estimates exactly 0
    lasso = Lasso(alpha =1.0).fit(X_train, y_train) / same as ridge
- **Elastic Net:** Tuning parameter – alpha1(def 1.0), alpha2
  If the l1_ratio/r(def 0.5, between 0, 1) is close to 0, it would weight the l2 penalty more (ridge), while if it is close to 1, it would weight the l1 penalty more (lasso).
  Ridge ->EN->Lasso

Linear Model for Classification:

- Cost function: -log(t) grows large when t approaches 0, cost will be large is model predicts close to 0 for positive class
- Uses maximum likelihood to estimate parameters
  *log_reg = LogisticRegression(penalty = None, random_state=42) / log_reg.classes_ / log_reg.coef_ / log_reg.intercept_*
  log_reg.predict_proba(X) / decision_boundary = X_new[y_proba[:, 1] >= 0.5][0, 0]
- Hyperparameter:
  - C: higher C, less regularization, every point matters more, default = 1.0
  - Penalty: l2 by def, can choose – l1, l2, EN or None
  - Logistic regression uses OvA by default but can set multi_class = "multinomial". **Auto** / over

- Solver = “lbfgs”, default

softmax_reg = LogisticRegression(C=30, random_state=42)

- **Multinomial Logistic / Softmax Regression:** Multiclass not multioutput,
- Cost function is cross entropy, equivalent to logistic regression with k=2.

clf = sk.linear_model.LogisticRegression(penalty = 'l2', multi_class = "multinomial", solver = 'saga', max_iter = 1000, tol = 0.1) / clf.predict(values, values)

Feature Selection and Preprocessing:

- Univariate Statistics:
  Anova test with score func:**f_classif** / f_regression->pick threshold for discarding features (p-value)
  select = SelectPercentile(percentile = 37.5) / select.fit(X) / select.transform(X_train) ***#k / percentile = 10*** *(int /all)*
- Model based feature selection: supervised learning like decision tree(feature_importances_) or coef_
  select = SelectFromModel(RandomForestClassifier(n_estimators=100, random_state=42), threshold='median')
- Iterative Feature Selection: Forward(SFS) / backward selection(RFE / SFS) until improvement
  select = RFE(RandomForestClassifier(n_estimators = 100, random_state = 42), n_features_to_select = 40)
  # n_features_to_select = None(half)
- **Preprocessing:**
  - Dummy data don’t need to be scaled
  - StandardScaler: ensure mean is 0 and variance is 1. Less affected by outliers. Z = (x-u) / s
  - *scaler = StandardScaler()*
  - MinMax scaler: Xs(max-min) + min. **(0,1)**
  - RobustScaler: remove median and scale data acc to interquartile range(25^{th} -75^{th}). Outliers are sample mean / variance
  - Normalizer: scale data point to have unit form. (text data / direction matters more)
- **Variable Encoding:**
  - **Categorical encoding:** converting text values to number. OrdinalEncoder() for converting, LabelEncoder() for output
  - **One Hot Encoding:** 0/1 for categorical variable. Handle_unknown = **“error”** / “ignore”
- **Imputation:** with zero, mean, median
  - **Simple Imputer: mean**, median, most_frequent(string & number)
  - KNNImputer: missing with mean of KNN values (numbers)
  - InterativeImputer: estimate each feature from all others(using regression) (numbers)
- **Column Transformer and Pipeline:**

preprocessing = ColumnTransformer(transformers=[("cat_col", OneHotEncoder(sparse_output=False, handle_unknown="ignore"), categorical_col), ("cont_col", StandardScaler(),Continuous_col)], remainder="passthrough") / preprocessing.fit_transform(X) / preprocessing.feature_features_name_out();
pipeline3 = Pipeline([("preprocess", preprocessing), ("log",LogisticRegression(tol=0.0005, C=0.1, solver='saga',max_iter=10000,penalty = 'l1', random_state=80))])
transformed_feature_names = pipeline_sfs.named_steps['log_reg'].get_feature_names_out()
cv1 = cross_val_score(pipeline3,X_train, y_binary_train, cv =kf,n_jobs=-1)
Note: pipeline.fit() - calls fit_transform() sequentially on all transformers, until last one where just fit().

Support Vector Machines

Can perform linear classification + non-linear classification with kernel trick / Regression / Outlier detection.
Works best with many features and not too many observations.

- **Separable Hyperplane:**

Sensitive to scaling and outliers – pipeline needed.

- **Linear Support Vector Classifier : Hard Margin / soft margin**

Linear SVC or SVC with linear kernel, Parametric

- Within the margin: $-1<=(w^Tx_i+b)<1$; $t_i(w^Tx_i+b)>=1$, outside margin and on correct side. Allows slack variable
  W, weight vector for margin; b, bias where located. Smaller w->larger margin(for x) and vice versa.
  $t_i(w^Tx_i+b)<1$, incur penalty when outside margin
  Hinge Loss: max(0, $t_i(w^Tx_i+b)$); $s = w^Tx_i+b$
  If $t_i = 1$ (positive class), then loss is 0 if $s_i \geq 1$
  If $t_i = -1$ (negative class), then loss is 0 if $s_i \leq -1$

- **Hyperparameter C** – smaller C, more regularization, less overfitting, wider street but more violation
  svm_clf = make_pipeline(StandardScaler(), LinearSVC(C=1, loss="hinge" random_state=42)
  svm_clf.predict(X_new)

If linearly not separable, can create a linearly separable dataset by adding polynomial features.

polynomial_svm_clf = make_pipeline(PolynomialFeatures(degree = 3), StandardScaler(), LinearSVC(C = 10, max_iter 10_000, random_state = 42)). ***#degreee = 2***

- **SVM Classifiers Type:**
  - LinearSVC(loss="hinge", C=1) / Loss function: **squared hinge** or hinge
    tol parameter – how precise you want your solution.
    OvA multiclass reduction
  - SVC (Kernel = "linear",C=1): hinge loss(def), OvO ;
    For hard margin classifier, C=float("inf")
- **SVM: Non-linear Classifier / Polynomial Kernel (can be applied to another model as well not just SVM)**
- Key hyperparameters:
  - d – degree of polynomial features (Increase if underfitting, decrease if overfitting)
  - r (coef0) - Controls influence of high-degree vs low-degree polynomials. (r * d)
  - C - soft margin hyperparameter.
  - Kernel: 'linear','poly', **'rbf'**, 'sigmoid', 'precomputed' or callable.
    poly100_kernel_svm_clf = make_pipeline(StandardScaler(),SVC(kernel="poly", degree=10, coef0=100, C=5))
- **Gaussian/RBF(Radial basis function) Kernel :** how much each feature resembles a particular landmark
- The choice of γ and C can impact the resulting decision boundary. Low C encourages larger margin, vice versa.
- γ: how far influence of a training example reaches (low γ --> far/ less overfitting, high γ --> close / more variance / potential overfitting)
  rbf_kernel_svm_clf = make_pipeline(StandardScaler(), SVC(kernel="rbf", gamma=5, C=0.001))

Note: LinearSVC is faster than SVC(kernel="linear"). If training set not too larger – Gaussian RBF.

Decision Tree

(Classification / Regression / Multioutput). Cannot extrapolate new aspects of data / Trees are typically not competitive with best supervised learning. No scaling needed.

- **Classification** and Regression Tree **(CART)** algorithm
  DecisionTreeClassifier / Classification Impurity Metrics:
  tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
  tree_clf.predict_proba / .predict
  *tree_clf.feature_importances_*
- Gini Impurity: slightly faster to compute and if 0, node is pure (as 1-prob of something showing)
- Entropy: approaches 0 when all data points belong to the same class
- The 2 might differ, as Gini may isolate most frequent class in own branch, but entropy may produce more balanced tree.
- Classification and **Regression** Tree**:** Minimizes MSE / MAE or other for regression instead of Gini / entropy.
  dt = sklearn.tree.DecisionTreeRegressor(max_depth=2)
- **Hyperparameters:**
  - By default, tree is built without impurity.
  - Pruning the tree, deleting unnecessary nodes (adjusting 1 of these is sufficient to prevent overfitting): Max_depth,max_features,max_leaf_nodes, min_samples_split, min_samples_leaf, min_weight_fraction_leaf
  - *Criterion* for "entropy", gini by default
- **Issues**: High variance, CART can produce different model for same data (unless random_state is set)
- Increasing min_* hyperparam and reducing max_* will regularize the model.
- Very sensitive to small variation in training data.

Ensembles

Group of predictors (classifier / regressors) works best if predictors are independent from one another, for making different errors.

- **Bagging:** Classification & Regression, can be used with any model. Scale well.
- **Bootstrapping** (sampling with replacement, 70%) **& Aggregating** (reduces bias and variance)
- Similar bias but lower variance than single model - Lower variance as variance mean is taken by dividing by n.
- Train B different bootstrapped training datasets on size m. Can train the model and make predictions in parallel using n_jobs=-1
- Hard Voting / soft voting: Soft voting gives high score as it gives higher votes to more confident votes.
- **Bagging Regression:** average all the prediction
- **Bagging Classification**:
  - Hard voting eg statistical mode – majority votes among all
  - soft voting: predict class with highest class probability avg over all classifiers); (**def**, if classifier has predict_proba()), predict_proba() or parameter voting, n_estimator=10(def)
- *Bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500, oob_score = True, max_samples=100, n_jobs=-1, random_state=42), # **n_estimator=10 def, max_samples = 1(all samples)***
  Bag_clf.score(X_test, y_test) #accuracy score
- Bagging has slightly higher bias than pasting. *(bootstrap = False for pasting, def True)*
- **Ensemble vs Decision Tree:** can have same number of errors but decision boundary is less irregular.

- **Random Forest:** Decision tree trained by bagging. Add extra randomness than bagging.
  - Does not require scaling, work well with heavy parameters.
  - In contrast to bagging, it searches for the best feature among a random subset of features when splitting a node. It de-correlates the bagged trees, trade higher bias for lower variance.
  - The random forest classifier is an ensemble classifier. It is hugely popular because it is simple, fast, and produces excellent classification accuracy.
  - Work less for high dimensional sparse dataset like text like NLP.
  - Can use gridsearch for hyperparameter tuning.
  - sqrt(p) samples considered by default. *#max_samples*
  - **RandomeForestClassifier():** hyperparameter like DTC and BaggingClassifier
    RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,n_jobs=-1, random_state=42). #convinience and optimized
    #or
    bag_clf=BaggingClassifier(DecisionTreeClassifier(max_features="sqrt",max_leaf_nodes=16),n_jobs=-1, random_state=42,n_estimators=500)
  - **RandomForestRegressor()**
  - **Feature_importances_:** how much the tree nodes that use the feature reduce the impurity. Weighted average, each weight equal to number of training samples.
  - **Tuning Random Forest hyperparameters:**
    - N_estimator >= 100 (def 100), more tree more robust
    - **Key hyperparameters**: Max_features default is sqrt (n features), n_features for regression. Smaller features reduce overfitting. Pre – pruning may help.

- **Gradient Boosting Machines:** Combines several weak learners into strong learner. Tries to fit the new predictors to the residual error made by previous predictors.
  - May take longer time to train (require careful parameter tuning)
  - GradientBoostingRegressor / Classifier
    GradientBoostingRegressor(learning_rate=1.0,n_estimators=3,max_depth=2, random_state=42)
  - **Hyperparameters**:
    - Learning_rate: def 0.1, Low – more trees, better generalization(shrinkage). High – stronger corrections
    - N_estimators: def 100, too high may lead to overfitting.
    - Max_depth: def 3, (usually 1-5, for smaller model / faster prediction)
    - N_iter_no_change: **None**. (too low (high) --> underfit (overfit))
  - To find optimal number of trees, can use early stopping:
    - Staged_predict() – returns prediction at each stage of training, 1,2 trees. Find the min.
    - Warm_start = True – keep existing trees when fit() is called.
    - Subsample = 0.25 – each tree is trained on 25% of the training instances randomly.
  - XGBoost, automatically takes care of early stopping – fast, scalable, portable.
  - Default loss function is squared error, can be set using loss parameter.