2025-08-16 00:51

tags :

- Machine Learning
- Hands on ML - book

Chapter 7 - Hands on

Voting Classifiers

- You train several different classifiers (e.g., logistic regression, SVM, random forest, k-nearest neighbors).
- Each model has similar performance (around 80% accuracy).
- Instead of relying on a single model, you **combine their predictions**.
- The final prediction is the **class chosen by majority vote** among the models.
- This technique is called a **hard voting classifier**, and it often performs better than the individual models alone.
- A **voting classifier** can often outperform the best individual classifier in the ensemble.
- Even weak learners (slightly better than random guessing) can combine into a strong learner if there are enough of them and they are diverse.
- Analogy: tossing a biased coin (51% heads). With many tosses, the probability of a majority of heads increases (law of large numbers). Similarly, many weak classifiers can yield high accuracy.
- **Limitation:** if classifiers are correlated (make similar mistakes), the benefit is reduced.
- **Key tip:** Ensemble methods work best when classifiers are **independent and diverse** (trained with different algorithms).

voting Classifier in Scikit-Learn

- **Class:** `VotingClassifier` (easy to use → give list of `(name, estimator)` pairs).
- **Behavior:** Clones and fits all estimators.
    - Original models → `.estimators`
    - Fitted clones → `.estimators_`
    - Dict access → `.named_estimators_`

```python
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
voting_clf = VotingClassifier( estimators=[
    ('lr',LogisticRegression(random_state=42)),
    ('rf', RandomForestClassifier(random_state=42)),
    ('svc', SVC(random_state=42)) ] )
voting_clf.fit(X_train, y_train)
```

Hard Voting

- Predicts class by **majority vote**.
- Example (moons dataset):
  - Logistic Regression = 86.4%
  - Random Forest = 89.6%
  - SVM = 89.6%
  - Voting Classifier = **91.2%** (better than all individuals).

Soft Voting

- Uses **average of predicted probabilities** → class with highest average prob.
- Usually performs better than hard voting.
- Requirement: all classifiers must support `predict_proba()`.
  - For SVM(not have `predict_proba()`), set `probability=True`.
    - this will make the SVC class use cross validation to estimate class probabilities, slowing down training, and it will add a predict_proba() method)
- Example: Soft voting improved accuracy to **92%**.

Key takeaway:

- Voting ensembles often outperform individual models, with **soft voting** usually being the best option if probability estimates are available.

Bagging and Pasting

- **Idea:** Use the same algorithm but train on **different random subsets** of the training set.
  - **Bagging (Bootstrap Aggregating):** sampling **with replacement**.
  - **Pasting:** sampling **without replacement**.
- **Sampling:**

    - Across predictors → instances can repeat.
    - Within a predictor → only Bagging allows repetition.
- **Prediction Aggregation**
    - **Classification:** mode (majority vote).
    - **Regression:** average.
- **Effect on Bias/Variance**
    - Each predictor individually → higher bias.
    - Aggregated ensemble →
        - **Bias ≈ single predictor(fitted on the all dataset)**
        - **Variance ↓ (reduced)** → better generalization.
- **Parallelism**
    - Training and prediction can be done **in parallel** (different cores/servers).
    - Makes Bagging & Pasting **scalable and efficient**.
- **Key takeaway:**
    - Bagging (with replacement) and Pasting (without replacement) create ensembles that reduce variance, often outperforming a single model, and are highly parallelizable.

Bagging and Pasting in Scikit-Learn

- **Classes:**
    - `BaggingClassifier` (classification)
    - `BaggingRegressor` (regression)
- **Key Parameters**
    - `n_estimators` : number of base models (e.g., 500 trees).
    - `max_samples` : number of training instances per base model (e.g., 100).
    - `bootstrap=True` → **Bagging** (with replacement).
    - `bootstrap=False` → **Pasting** (without replacement).
    - `n_jobs=-1` → use all CPU cores (parallel training/prediction).
- Code

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
bag_clf = BaggingClassifier(DecisionTreeClassifier(),
 n_estimators=500, max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

- **Soft Voting**
    - If base estimator supports `predict_proba()` (e.g., decision trees), ensemble uses

soft voting automatically.

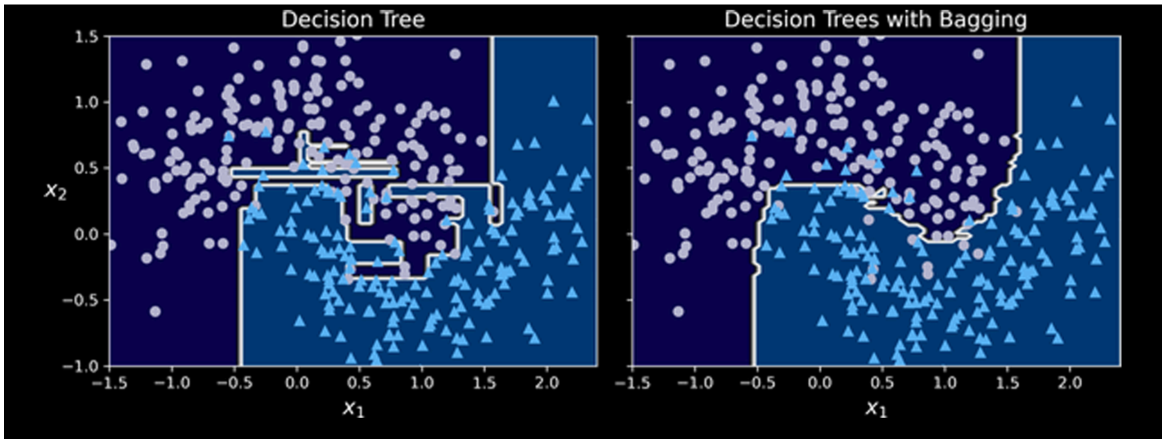
- **Bias–Variance Effect**
  - Bagging & Pasting → reduce **variance** compared to a single model.
  - Bagging → slightly higher bias than pasting (more randomness), but **lower correlation** → usually better performance.
  - **Generalization:** Ensemble smoother decision boundaries, less overfitting than a single decision tree.
  - 

    Figure 7-5. A single decision tree (left) versus a bagging ensemble of 500 trees (right)
- **Practical Tip**
  - Bagging usually preferred.
  - But test both (via cross-validation) if resources allow.
- **Key takeaway:**
  `BaggingClassifier` (or `Regressor`) = train multiple predictors on random subsets of data. Bagging (with replacement) often beats pasting due to reduced correlation, giving better generalization.

Out-of-Bag Evaluation

- In **Bagging**:
  - Each predictor is trained on a bootstrap sample (size = training set).
  - On average → ~63% of training instances are sampled.
  - Remaining ~37% = **Out-of-Bag (OOB) instances** (different for each predictor).
- **Use of OOB Instances**
  - Serve as a **built-in validation set** → no need for a separate one.
  - Each training instance is OOB for some predictors → use those to get ensemble predictions.
  - Compute accuracy (or other metrics) based on OOB predictions.
- **In Scikit-Learn**
  - Enable with `oob_score=True`.
  - Accuracy available via `.oob_score_`.

- Class probability estimates available via `.oob_decision_function_`.

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(),
        n_estimators=500, ... oob_score=True,
        n_jobs=-1, random_state=42)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_ # 0.896


>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred) # 0.92
```

- **Example Results**
  - OOB score ≈ **89.6%**.
  - Actual test set accuracy = **92%** (OOB slightly pessimistic, ~2% lower).
  - `oob_decision_function_` → gives predicted class probabilities per instance.
- **Key takeaway:**
  OOB evaluation provides a **free, internal validation estimate** for bagging models—usually close to test accuracy, but slightly pessimistic.

Random Patches and Random Subspaces

- **Feature Sampling in Bagging**
  - Controlled by:
    - `max_features` → number/ratio of features to sample.
    - `bootstrap_features` → whether sampling is **with replacement**.
  - Works like `max_samples` & `bootstrap`, but for **features** instead of instances.
- **Methods**
  1. **Random Patches**
     - Sample both **instances** and **features**.
     - Default bagging + feature sampling.
  2. **Random Subspaces**
     - Keep **all training instances** (`bootstrap=False`, `max_samples=1.0`).
     - Sample only **features** (`bootstrap_features=True` and/or `max_features < 1.0`).
- **Effect**
  - Increases **diversity** among predictors.
  - Leads to **lower variance**, but adds a bit more **bias**.
  - Particularly useful for **high-dimensional data** (e.g., images) → speeds up training.

- **Key takeaway:**
  Feature sampling (random patches/subspaces) enhances diversity and reduces variance, at the cost of slightly higher bias—especially helpful in high-dimensional problems.

Random Forests

- **Random Forests**
  - **Definition:** Ensemble of decision trees, usually trained with **bagging** (sometimes pasting).
  - **Classes:**
    - `RandomForestClassifier` (classification)
    - `RandomForestRegressor` (regression)
  - **Convenience:** Has all hyperparameters of
    - `DecisionTreeClassifier` (tree growth control)
    - `BaggingClassifier` (ensemble control).
- **Extra Randomness**
  - At each split, instead of checking all features → use a **random subset**.
  - Default = √n features (classification).
  - Increases **diversity** among trees.
  - Effect:
    - Slightly ↑ bias
    - ↓ variance
    - Better overall generalization.
- **Equivalence**

  ```
  RandomForestClassifier(n_estimators=500, max_leaf_nodes=16)
  ```

  ≈

  ```
  BaggingClassifier(     DecisionTreeClassifier(max_features="sqrt",
  max_leaf_nodes=16),     n_estimators=500 )
  ```

- **Key takeaway:**
  Random Forests = bagging of decision trees + **feature randomness** at splits → strong, low-variance models with good generalization.

Extra-Trees

- Extra-Trees (Extremely Randomized Trees)

- In **Random Forests**, each split considers a random subset of features.
- In **Extra-Trees**, extra randomness is added:
  - **Random thresholds** are chosen for splits (instead of searching for the best threshold).
- **Trade-off**:
  - ✅ Lower variance
  - ✅ Faster training (no need to search best thresholds)
  - ❌ Higher bias
- **Implementation in Scikit-Learn**:
  - `ExtraTreesClassifier` (for classification)
  - `ExtraTreesRegressor` (for regression)
  - API is the same as RandomForest classes, but **bootstrap=False by default**.
- **Key Point**: No way to know in advance if Random Forests or Extra-Trees work better → must test both with **cross-validation**.

Feature Importance

- Feature Importance in Random Forests
  - **Definition**: Importance of a feature is based on how much it reduces impurity across all trees.
  - **Weighting**: Each node's contribution is weighted by the number of training samples it affects.
  - **Scaling**: Importances are normalized so they sum to 1.
  - **Access**: Available through `feature_importances_` after training.
- **Example (Iris dataset):**

```python
from sklearn.datasets import load_iris
iris = load_iris(as_frame=True)
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris.data, iris.target)
for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
    print(round(score, 2), name)
```

  - Sepal length → **0.11**
  - Sepal width → **0.02**
  - Petal length → **0.44**
  - Petal width → **0.42**
- Key Uses:

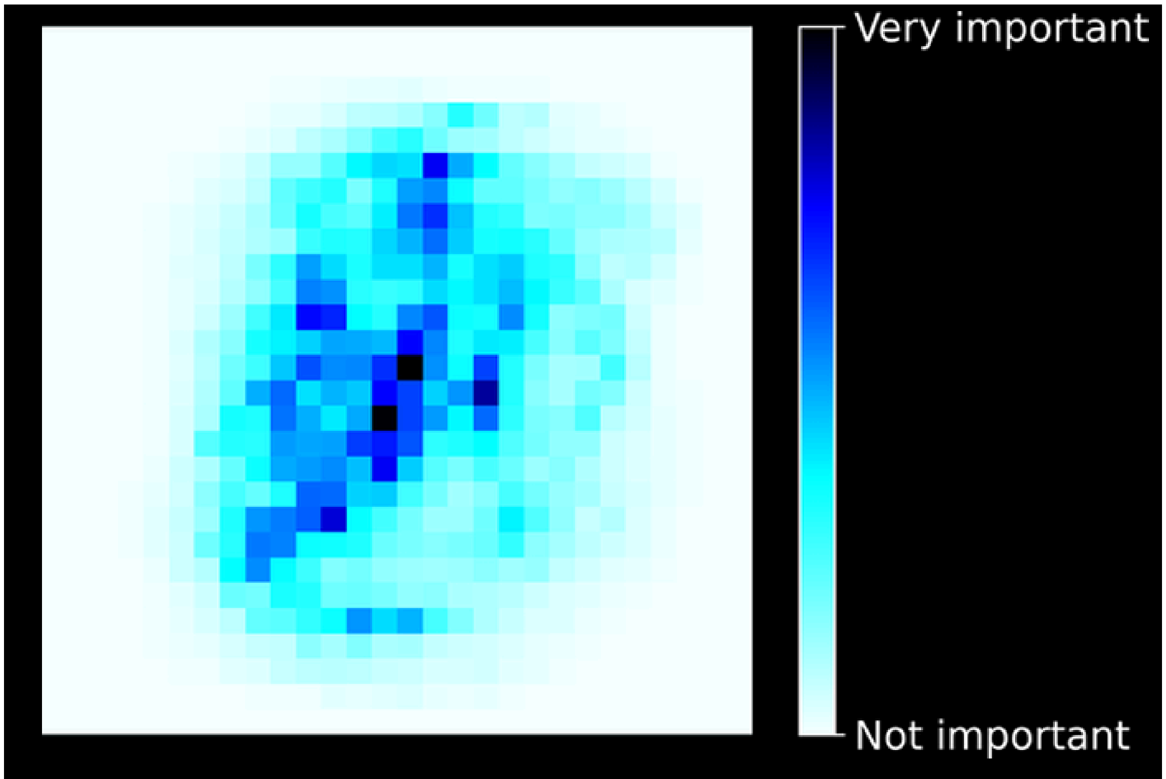
    - Helps **interpret models** (which features matter most).
    - Useful for **feature selection**.
    - Can visualize importance (e.g., MNIST pixel map).
        - 

Figure 7-6. MNIST pixel importance (according to a random forest classifier)

Boosting

- **Definition**: Ensemble method that combines several **weak learners** into a **strong learner**.
- **Key Idea**: Learners are trained **sequentially**, each new model focuses on correcting the errors of the previous one.
- **Popular Methods**:
    - **AdaBoost** (Adaptive Boosting)
    - **Gradient Boosting**
- Purpose: Improve accuracy by reducing bias and variance compared to single weak learners.

AdaBoost

- Core idea
    - Build an ensemble **sequentially**: each new weak learner focuses more on **instances misclassified** by previous learners (by **increasing their weights**).
    - Final prediction is a **weighted vote** of all learners.
- Training procedure (binary or multiclass via SAMME)

- **Initialization**
  - For mmm training instances, set **instance weights**:
    $w_i^{(1)} = \dfrac{1}{m}$ for $i = 1, \ldots, m.$
- **For** $j = 1 \ldots N$ **(number of estimators):**
  1. **Fit** base learner h_j on the training set **using current weights** $\{w_i^{(j)}\}$.
  2. **Weighted error**:
     $r_j = \sum_{i=1}^{m} w_i^{(j)} \cdot \mathbf{1}[h_j(x_i) \neq y_i]$
  3. **Learner weight (vote)**:
     $\alpha_j = \eta \cdot \log\left(\frac{1-r_j}{r_j}\right)$
     - η = **learning rate** (default 1.0).
     - If $r_j < 0.5 \to \alpha_j > 0$ (useful learner).
     - If r_j = 0.5 $\to \alpha_j = 0$ (no influence).
     - If $r_j > 0.5 \!\to \alpha_j < 0$ (worse than random).
  4. **Update instance weights** :
     $w_i^{(j+1)} = \begin{cases} w_i^{(j)} & \text{if } h_j(x_i) = y_i \\ w_i^{(j)} \cdot \exp(\alpha_j) & \text{if } h_j(x_i) \neq y_i \end{cases}$
     - Then **normalize**: $w_i^{(j+1)} \leftarrow \dfrac{w_i^{(j+1)}}{\sum_{k=1}^{m} w_k^{(j+1)}}$
  5. **Stop** when N learners are trained or a **perfect** learner is reached.
     Intuition: Misclassified points **gain weight** by a factor $\exp(\alpha_j)$, pulling the next learner to focus on the "hard" cases. Smaller η (learning rate) → smaller weight boosts each round (smoother learning, often better generalization).
- Prediction rule
  - **Weighted majority vote** (SAMME):
    $\hat{y}(x) = \arg\max_k \sum_{j=1}^{N} \alpha_j \cdot \mathbf{1}[h_j(x) = k]$
  - **SAMME.R** (if base learners output probabilities): uses **class probabilities** instead of hard labels; generally performs **better**.
- Relation to gradient descent
  - Conceptually similar: instead of tweaking one model's parameters to minimize loss, AdaBoost **adds learners** sequentially, each step reducing the ensemble's error (see the "decision boundaries" plots: halving the learning rate → gentler updates, smoother boundaries).
- Parallelism & scaling
  - **Drawback:** Training is **sequential** (each round depends on the previous), so it **cannot be parallelized** like bagging/pasting.
- Scikit-Learn essentials
  - **Classes:** `AdaBoostClassifier`, `AdaBoostRegressor`.

- **Default base estimator: decision stump** `DecisionTreeClassifier(max_depth=1)`.
- **Common params:**
    - `n_estimators` → number of weak learners NNN
    - `learning_rate` → η\etaη (shrinks αj\alpha_jαj)
- **Code pattern:**

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
ada_clf = AdaBoostClassifier(
            DecisionTreeClassifier(max_depth=1),
            n_estimators=30,
            learning_rate=0.5,
            random_state=42 )
ada_clf.fit(X_train, y_train)
```

- **Multiclass:** uses **SAMME**; if base learners support `predict_proba`, Scikit-Learn can use **SAMME.R** (usually better).

- Practical tips
    - If **overfitting**:
        - Decrease `n_estimators`, and/or
        - **Regularize** the base learner more (e.g., shallower trees, min samples split/leaf).
    - Use smaller `learning_rate` with larger `n_estimators` for a stronger but smoother ensemble.
- Ultra-short flashcard
    - **Updates**: $\alpha_j = \eta \log \frac{1-r_j}{r_j}$, miscls weights × e^{α_j}, then **normalize**.
    - **Predict**: weighted vote by αj\alpha_jαj.
    - **Pros**: strong from weak learners; **Cons**: sequential (no parallel).
    - **SAMME / SAMME.R** for multiclass; **stumps** are common base learners.

Gradient Boosting(GBRT)

- Core idea
    - Like AdaBoost, builds ensemble **sequentially**.
    - Instead of reweighting misclassified samples, each new predictor is trained to **fit the residual errors** of the previous predictors.
    - Base learners: usually **decision trees** → Gradient Boosted Regression Trees (GBRT).
- Step-by-step (regression example)
    1. Fit first tree $h_1(x)$ to data.

2. Compute residuals $r_1 = y - h_1(x)$.
3. Fit second tree $h_2(x)$ to residuals.
4. Compute new residuals $r_2 = r_1 - h_2(x)$.
5. Continue for N trees.

- **Final prediction:**
  $\hat{y}(x) = \sum_{j=1}^{N} h_j(x)$
- Scikit-Learn implementation

  ```python
  from sklearn.ensemble import GradientBoostingRegressor
  gbrt = GradientBoostingRegressor(
              max_depth=2,
              n_estimators=3,
              learning_rate=1.0,
              random_state=42 )
  gbrt.fit(X, y)
  ```

  - `GradientBoostingRegressor` (for regression)
  - `GradientBoostingClassifier` (for classification)
- Key hyperparameters
  - **Tree growth controls:** `max_depth` , `min_samples_leaf` , etc.
  - **Ensemble controls:**
    - `n_estimators` : number of trees
    - `learning_rate` : scales contribution of each tree (smaller → need more trees, but better generalization → called **shrinkage**)
    - `subsample` : fraction of training instances per tree → stochastic gradient boosting (adds randomness, reduces variance, speeds training)
- Regularization & Overfitting control
  - **Shrinkage (learning_rate):**
    - Lower `learning_rate` + higher `n_estimators` → better generalization.
  - **Early stopping:**
    - `n_iter_no_change` : stop if no improvement for k iterations.
    - Uses an internal validation set (`validation_fraction` , default 10%).
    - `tol` : threshold for negligible improvement.
  - **Subsampling:**
    - Train each tree on a subset of training set (e.g., `subsample=0.25`) → higher bias, lower variance, faster training.
- Practical tips
  - Use **grid/randomized search** to find `n_estimators` and `learning_rate` .

- Safer default: small `learning_rate` (0.05–0.1) with large `n_estimators`.
- Early stopping (`n_iter_no_change`) helps pick optimal number of trees automatically.
- For very large datasets: consider stochastic gradient boosting (`subsample < 1.0`).

- Flashcard summary
  - **Trains on residuals**, not weights (unlike AdaBoost).
  - Prediction = sum of all trees.
  - **Shrinkage** = small learning_rate + more trees.
  - **Regularization:** early stopping, subsampling.
  - **Tradeoff:** more trees → better fit, but risk of overfitting.

Histogram-Based Gradient Boosting

- **What it is**:
  An optimized implementation of Gradient Boosted Regression Trees (GBRT) in Scikit-Learn, designed for **large datasets**.
- **How it works**:
  - Bins input features into integers (`max_bins` hyperparameter, default = 255, max = 255).
  - Binning reduces the number of thresholds to evaluate.
  - Using integers → faster & memory-efficient data structures.
  - Removes need to **sort features** when training each tree.
- **Complexity**:
  - Standard GBRT: **O(n × m × log(m))**
  - HGB: **O(b × m)**
    - `n` : number of features
    - `m` : number of instances
    - `b` : number of bins
  - ⇒ HGB can be **hundreds of times faster** on large datasets.
- **Trade-off**:
  - Binning causes **precision loss**.
  - Acts as a **regularizer** → can reduce overfitting or may cause underfitting (dataset dependent).
- Classes in Scikit-Learn:
  - `HistGradientBoostingRegressor`
  - `HistGradientBoostingClassifier`
    (similar to `GradientBoostingRegressor` / `Classifier`, with key differences).
- Key Differences from Standard GBRT:
  1. **Early Stopping**

        - Auto-activated if dataset > 10,000 instances.
        - Can force with `early_stopping=True/False`.
    2. **Subsampling**
        - **Not supported** in HGB.
    3. **n_estimators → renamed to max_iter**
    4. **Tree hyperparameters allowed**:
        - `max_leaf_nodes`
        - `min_samples_leaf`
        - `max_depth`
    5. **Supports categorical features & missing values**
    - No need for imputer, scaler, or one-hot encoder.
    - **Categorical features must be integers** (0 to `< max_bins`).
    - Use `OrdinalEncoder` for transformation.
- Example Pipeline (California Housing Dataset)

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder
hgb_reg = make_pipeline(
            make_column_transformer(
                (OrdinalEncoder(), ["ocean_proximity"]),
                remainder="passthrough"
                ),
            HistGradientBoostingRegressor(categorical_features=[0],
            random_state=42) )
hgb_reg.fit(housing, housing_labels)
```

    - Very compact pipeline.
    - No imputer, scaler, or one-hot encoder required.
    - Without tuning → **RMSE ≈ 47,600**.
- Extra Note (TIP)
    - Other **optimized gradient boosting libraries**:
        - **XGBoost, CatBoost, LightGBM** → mature, GPU support, many features.
        - **TensorFlow Random Forests** → optimized implementations for random forests, Extra-Trees, GBRT, etc.

Stacking

- **Idea**: Instead of using simple aggregation (like voting), train a **model (meta-learner or blender)** to combine the predictions of base models.
- **Process**:
    1. Train base predictors.
    2. Use **cross_val_predict()** to generate out-of-sample predictions from each base model.
    3. These predictions form a **new training set** (each model gives 1 feature).
    4. Train the blender (final estimator) on this new dataset with the original targets.
    5. Retrain base models on the full dataset and use them + blender for final prediction.
- **Multi-layer stacking**:
    - Possible to train multiple blenders (different algorithms).
    - Outputs can feed into another blender (multi-layer stacking).
    - Can slightly improve performance but increases complexity and training cost.
- **Scikit-Learn Implementation**
    - Classes:
        - `StackingClassifier`
        - `StackingRegressor`
    - Example:

    ```
    from sklearn.ensemble import StackingClassifier
    stacking_clf = StackingClassifier(
                    estimators=[
                        ('lr', LogisticRegression(random_state=42)),
                        ('rf',RandomForestClassifier(random_state=42)),
                        ('svc', SVC(probability=True, random_state=42))
                    ],

    final_estimator=RandomForestClassifier(random_state=43),
                    cv=5)
    stacking_clf.fit(X_train, y_train)
    ```

    - Behavior:
        - Uses `predict_proba()` if available, else `decision_function()`, else `predict()`.
        - Default final estimator:
            - `LogisticRegression` for classifiers.
            - `RidgeCV` for regressors.
    - **Performance & Takeaways**
    - Stacking often improves accuracy (example: **92.8% vs 92% with soft voting**).
    - **Best practices**:

- Try **Random Forests, AdaBoost, Gradient Boosted Trees** first (strong, versatile).
- Ensembles are especially good for **heterogeneous tabular data**.
- Require **little preprocessing**, making them great for fast prototyping.
- Voting + Stacking can help **squeeze maximum performance**.

Resources :

-

Related notes :

-

References :

- ### Internal :
  -
  -
  -
- ### External :
  - hegab videos
  - the book
  - the notebook
  -