Pivot table

Sns.barplot

**Cross\_val\_score**

* By default, the score computed at each CV iteration is the score method of the estimator. It is possible to change this by using the scoring parameter.

**n\_jobs**:  is used to specify how many concurrent processes or threads should be used.  If 1 is given, no joblib parallelism is used at all, which is useful for debugging. If set to -1, all CPUs are used. For n\_jobs below -1, **(n\_cpus + 1 + n\_jobs)** are used.

**verbose**: verbose = 0: showing nothing about the progress (silent), verbose > 1: more details being printed (e.g. “Fitting 3 folds for each of 8 candidates, totalling 24 fits“)

**scoring**: controls what metric GridSearchCV or cross\_val\_score apply to the estimators evaluated. A list of common cases is listed here: [3.3. Metrics and scoring: quantifying the quality of predictions — scikit-learn 1.3.2 documentation](https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter)

**VotingClassifier**

**StandardScaler**: [How to use StandardScaler in Pandas? - Python Tutorial - YouTube](https://www.youtube.com/watch?v=ZMEy9kqG1Ec&list=PLVO0f8uddRamtXCPF1Oxxf6PrQ0beIb44&index=10&t=2s)

**stats.norm.fit / mu, sigma / stats.norm.pdf:**

[A Gentle Introduction to Probability Density Estimation - MachineLearningMastery.com](https://machinelearningmastery.com/probability-density-estimation/)

[Cumulative Distribution Functions and Probability Density Functions - YouTube](https://www.youtube.com/watch?v=3xAIWiTJCvE)

* mu: mean value, sigma: standard deviation
* Probability Density function (PDF): the calculation of probabilities for specific outcomes of a random variable. The shape of the probability density function across the domain for a random variable is referred to as the probability distribution and common probability distributions have names, such as *uniform, normal, exponential,* and so on.
* Q-Q plot (stats.probplot) / qq\_data[0][0] and qq\_data[0][1]
* Stats.linregress
* Ax.bar\_label -> label\_type
* SimpleImputer -> strategy, fill\_value
* OneHotEncoder -> handle\_unknown, sparse
* Param\_grids: A screen shot of a computer

  Description automatically generated
* KFold vs StratifiedKFold

**XGBoost (Extreme Gradient Boost)**: can do both regression and classification. It contains

* Gradient Boost
* Regularization
* Unique Regression Tree
* Parallel Learning

**LightGBM** as lgb:

* Steps:

1. X\_train\_data = lgb.Dataset(X\_train, label=y\_train)

X\_test\_data = lgb.Dataset(X\_test, label=y\_test)

1. Set “Parameters” dictionary as *param*.
2. model\_lgb = lgb.train(param, X\_train\_data, num\_boost\_round=*int*, valid\_sets=X\_test\_data)
3. y\_pred = model\_lgb.predict(**X\_test**, num\_iteration=model\_lgb.best\_iteration)

* Parameters:
  + ‘objective’: ‘binary’ 🡺 we want to predict “0” and “1” targets.
  + ‘metric’: ‘auc’ 🡺 metric of validation
  + ‘is\_unbalance’: ‘true’ 🡺 used when the training data is not split equally between “0” and “1”
  + ‘boost’: ‘gdbt’ 🡺 choose boosting type, in this case: *gradient boosting*
  + ‘num\_leaves’: *int* 🡺 define number of leaves nodes in each tree
  + ‘feature\_fraction’: *float* 🡺 every tree that LightGBM creates will have only **(*float*\*100) %** of the features
  + ‘bagging\_fraction’: *float* 🡺 only **(*float*\*100) %** of the observation records will be used
  + ‘bagging\_freq’: *int* 🡺 after every *int* iterations, LightGBM will randomly pick **(*float*\*100) %** of the observation records and use them for the next *int* iterations
  + ‘learning\_rate’: *float* 🡺 scaling for each time a tree is added to prediction
* lgb.train():
  + num\_boost\_round: *int* 🡺 number of trees

**GridSearchCV**: is the process of performing hyperparameter tuning in order to determine the optimal values for a given model. There is no way to know in advance the best values for hyperparameters so ideally, we need to try all possible values to know the optimal values.

* **How does GridSearchCV calculate its best\_score\_?**

“Mean cross-validated score of the best\_estimator”

(e.g. If cv=5, the data will be split into train and test folds 5 times. The model will be fitted on *training sets* and scored on the *testing sets*. These 5 test scores are averaged to get the score. The highest score will be the *best\_score\_)*

* FunctionTransformer
* ColumnTransfromer -> remainder=’passthrough’
* Why we need np.exp() when predicting y\_pred?

A screenshot of a computer

Description automatically generated