

Week1

- Big picture
 - What are the inputs and outputs? Single or multiple?
 - What's the business objective?
 - Algo selection – Supervised-Unsupervised-RL, Regression-Classification, Continuous learning-periodic updates, Batch-Online
 - Select performance measure – Regression (MSE-MAE), Classification (Precision-Recall-F1-Score-Accuracy)
 - What is the current solution? Provides for a useful baseline
- Get the data
 - Spread about multiple tables, files, documents available locally, or on the web.
 - Structured-Unstructured data.
 - Understand the data and its statistics – `info()` and `describe()`
 - Distribution of examples by features/label. Use a histogram.
 - Create test set using `train_test_split()` in order to avoid data snooping bias. Used to select k% data for test-set
 - `train_test_split()` may be used to process multiple datasets with an identical number of rows and select the same indices from these datasets. Useful when labels are in different dataframes.
 - Alternatively, use stratified sampling to divide the population into homogeneous groups called strata and sample data from each strata representative of overall distribution. Use `StratifiedShuffleSplit.split()` to divide data between train and test.

• • •

```
1 from sklearn.model_selection import StratifiedShuffleSplit
2 split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
3 for train_index, test_index in split.split(data, data["quality"]):
4     strat_train_set = data.loc[train_index]
5     strat_test_set = data.loc[test_index]
```

- Visualize the data.
 - Enables to understand features and their relationship among themselves and with the output label.
 - Commonly used metric is standard correlation coefficient $\{-1, 1\}$ to measure correlation among features.

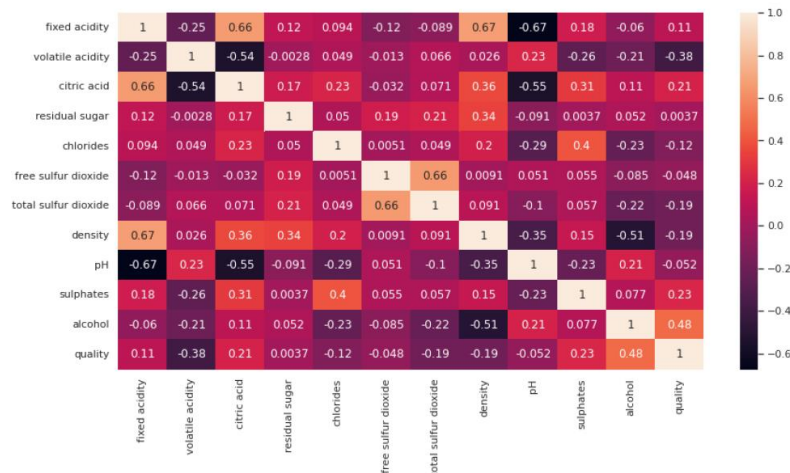
• • •

```
1 corr_matrix = exploration_set.corr()
```

- Correlation coefficient can capture only linear relationship; use rank correlation, for non-linear relationship.
- Use heatmap to visualize correlation matrix.

• • •

```
1 plt.figure(figsize=(14,7))
2 sns.heatmap(corr_matrix, annot=True)
```



- Note that correlation coefficient on the diagonal is 1.
 - Darker colors (black) are used to represent negative correlations, while fainter colors are used to denote positive correlations.
 - Alternatively use a scatter matrix to visualize correlation.
- Pre-processing data
 - Problems with data includes outliers, missing values, different scales, non-numeric attributes, non-amenable distribution of data.
 - Use `isna().sum()` to find out if there are missing values. In case there are missing values, use `SimpleImputer` (with appropriate strategy like mean-median) or `KNNImputer` to fill them up. However, make sure that non-numerical values are dropped before applying imputing step.

```

from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")

imputer.fit(wine_features)
tr_features = imputer.transform(wine_features)

```

 - Alternatively, you can drop the corresponding row using `dropna()` or `drop()`
 - Convert categories to numbers using `OrdinalEncoder` or `OneHotEncoder`
 - Use `MinMaxScaler` or `StandardScaler` for feature scaling.
 - Scaling techniques should be *learned* on the training data, and can be used to transform the training and test data.
 - Pipelines are typically used to perform operations sequentially. The pipeline exposes the same method as the final estimator.

```

1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import StandardScaler
3 transform_pipeline = Pipeline([
4     ('imputer', SimpleImputer(strategy="median")),
5     ('std_scaler', StandardScaler()),
6 ])
wine_features_tr = transform_pipeline.fit_transform(wine_features)

```

 - Here, the pipeline first performs imputation of missing values using `SimpleImputer` (with a 'median' strategy) and its result is passed for standardization.
 - Pipelines only work with one type of data. Use `ColumnTransformer` to work with columns of different types.
- Select and train ML model

- Build a quick base-line model. Evaluate the performance on training as well as test sets, using `mean_squared_error()`, `mean_absolute_error()`, `r2_score()` etc.
- We can use a cross-validation for robust performance of model performance, where multiple validation sets are created from the training set. This process generates a separate MSE for each validation set. We can use the mean/std.deviation of all such MSEs to evaluate the performance.
- It's a good practice to build a few quick models without tuning hyperparameters and shortlist a few promising models among them, and work on them further.
- In case of underfitting, use model with more capacity and use less constraints/regularization.
- In case of overfitting, use more data, a simpler model, and use more constraints/regularization.
- In order to find the best combination of hyper-parameters, use `GridSearchCV` or `RandomizedSearchCV`, both of which build multiple models with different sets of hyper-parameters.
- Utilize the `param_grid` to work with all combinations of these parameter values. For example, in order to select the best combination for a `RandomForest` regressor, you may use the following `param_grid`.

```

1 param_grid = [
2   {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
3   {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
4 ]

```

In the above case, the first dictionary yields 12 combinations and second dictionary yields 6 combinations, thus resulting in 18 total combinations.

- The models could be built in parallel, by setting parameter `n_jobs` to -1.
- Set parameter `error_score` to 0, if you do not want the process to stop when one of the models fails to build for any reason.
- Find the best parameter set and the best model resulting from the process using `best_params_` and `best_estimator_` attributes respectively.
- Setting parameter `refit` to True, retrains the best estimator (found using `best_estimator_` property) on the full training set, so as to improve upon the model again. Note that the original test set is not used in the process.
- `GridSearchCV` can prove costly when the hyperparameter space is quite large. In such cases, use `RandomizedSearchCV`.
- `RandomizedSearchCV` selects a random value for each hyperparameter at the start of each iteration and repeats the process for `n_iter` random combinations.
- In the case of `GridSearchCV/RandomizedSearchCV`, `best_estimator_.feature_importances_` property allows to selectively drop features based on their importances.
- Apart from the above two generic cross-validation methods, certain models have more specialized methods for cross-validation. Examples include `LassoCV`, `RidgeCV`, `ElasticNetCV`
- Once the predictions are made on the test set, it's a good idea to get 95% confidence interval of the evaluation metric.

```

1 from scipy import stats
2 confidence = 0.95
3 squared_errors = (quality_test_predictions - wine_labels_test) ** 2
4 stats.t.interval(confidence, len(squared_errors) - 1,
5                  loc=squared_errors.mean(),
6                  scale=stats.sem(squared_errors))
(0.29159276569581916, 0.4153100120819586)

```

- Present your solution
 - Document everything
 - Create clear visualizations.
- Launch, monitor and maintain your system
 - System outages
 - Degradation of model performance.
 - Manual evaluations
 - Regular assessments on data quality

Data loading mechanisms

- The dataset loaders (load_*) are used to load toy datasets bundled with sklearn.
- The dataset fetchers (fetch_*) are used to download and load datasets from the internet.
- The dataset generators (make_*) are used to generate controlled synthetic datasets.
- Both loaders and fetchers return a Bunch object, which is a dictionary with two keys data and target. There could be additional keys like feature_names, target_names, DESCR and filename
- Generators return a tuple containing the data and target.
- Loaders and fetchers can also return tuple (like in the case of generators), if return_X_y is set to True.
- make_regression() produces regression datasets.

```
X, y = make_regression(n_samples=100, n_features=5, n_targets=1, shuffle=True, random_state=42) #
```

generates single label regression data.

```
X, y = make_regression(n_samples=100, n_features=5, n_targets=5, shuffle=True, random_state=42) #
```

generates multiple label regression data

- make_blobs() and make_classification() creates a specified number of normally-distributed clusters of points and then assign one or more clusters to each class thereby creating multi-class datasets.

```
X, y = make_blobs(n_samples=10, centers=3, n_features=2, random_state=42) # Generates 3 clusters.
```

Used in unsupervised ML

```
X, y = make_classification(n_samples=100, n_features=10, n_classes=2, n_clusters_per_class=1, random_state=42) #
```

Generates multiple classes. Also generates clusters per classes.

- make_multilabel_classification() generates multi-label datasets.

```
X, y = make_multilabel_classification(n_samples=100, n_features=20, n_classes=5, n_labels=2)
```

Note that n_labels represents the average number of labels in each example.

Week2

- Typically pre-processing includes dealing with
 - missing values in features, (`sklearn.impute`)
 - scaling features (`sklearn.preprocessing`)
 - converting categorial features to numerical representation (`sklearn.preprocessing`)
 - extracting features from non-numeric data (`sklearn.feature_extraction`)
 - reducing too many features (`sklearn.decomposition.pca`)
 - expanding the features (`sklearn.kernel_approximation`)
- It's important to apply the same pre-processing method to the train, test and eval sets. Hence, it's best to use pipelines with appropriate transformations.

Feature extraction

- APIs typically used for feature extraction from data are
 - `DictVectorizer` # Converts original data in dictionary format to a matrix.

```
data =  
[{'age': 4, 'height': 96.0},  
 {'age': 1, 'height': 73.9},  
 {'age': 3, 'height': 88.9},  
 {'age': 2, 'height': 81.6}]
```

Example: `DictVectorizer` transforms into

4	96.0
1	73.9
3	88.9
2	81.6

- `FeatureHasher` # Outputs a sparse matrix, by applying a hash function to the features to determine the column index in the resultant matrix. `FeatureHasher` is a high-speed and low-memory vectorizer
- In addition, features can also be extracted from images and text using APIs built into the same module.

Imputing data

- `SimpleImputer` – uses one of the strategies *mean*, *median*, *most_frequent*, *constant* to impute

7	1
nan	8
2	nan
9	6

Example: `SimpleImputer` with a mean strategy transforms into

7	1
6	8
2	5
9	6

- `KNNImputer` – Find `n_neighbors` nearest neighbors based on Euclidean distance and uses mean on these datapoints to impute.

In the presence of missing coordinates, the Euclidean distance is calculated by ignoring the missing values and scaling up the weight of the non-missing coordinates.

$$d_{xy} = \sqrt{\text{weight} * \text{squared distance from present coordinates}}$$

where,

$$\text{weight} = \frac{\text{Total number of coordinates}}{\text{Number of present coordinates}}$$

For example, the Euclidean distances between two points (3, NA, 5) and (1, 0, 0) is:

$$\sqrt{\frac{3}{2} \{(3-1)^2 + (5-0)^2\}} = 6.595453$$

1.	2.	nan.
3.	4.	3.
nan	6.	5.
8.	8.	7.

1.	2.	4.
3.	4.	3.
5.5	6.	5.
8.	8.	7.

Example: `KNNImputer` with 2 neighbors transforms into

- While imputing, presence of missing values can be indicated by using the `MissingIndicator` API

Feature scaling

- APIs used are

- StandardScaler # $x' = \frac{x - \mu}{\sigma}$. Example: StandardScaler Transforms $\begin{bmatrix} 4 \\ 3 \\ 2 \\ 5 \\ 6 \end{bmatrix}$ into $\begin{bmatrix} 0 \\ -1/\sqrt{2} \\ -2/\sqrt{2} \\ 1/\sqrt{2} \\ 2/\sqrt{2} \end{bmatrix}$
- MaxAbsScaler # $x' = \frac{x - x.min}{x.max - x.min}$. Example: MaxAbsScaler Transforms $\begin{bmatrix} 15 \\ 2 \\ 5 \\ -2 \\ -5 \end{bmatrix}$ into $\begin{bmatrix} 1 \\ 0.35 \\ 0.5 \\ 0.6 \\ 0 \end{bmatrix}$
- MinMaxScaler # $x' = \frac{x}{\text{MaxAbsoluteValue}}$, where $\text{MaxAbsoluteValue} = \max(x.max, |x.min|)$
Note that the resultant values fall within range [-1,1]

Example: MinMaxScaler transforms $\begin{bmatrix} 4 \\ 2 \\ 5 \\ -2 \\ -100 \end{bmatrix}$ into $\begin{bmatrix} 0.04 \\ 0.02 \\ 0.05 \\ -0.02 \\ -1 \end{bmatrix}$

- MinMaxScaler fits every feature into a range [0, 1]
- StandardScaler is less affected by outliers than MinMaxScaler.

Other transformers

- FunctionTransformer applies user defined function on dataset in order to transform.

Example: log2 transforms $\begin{bmatrix} 128 & 2 \\ 2 & 256 \\ 4 & 1 \\ 512 & 64 \end{bmatrix}$ into $\begin{bmatrix} 7 & 1 \\ 1 & 8 \\ 2 & 0 \\ 9 & 6 \end{bmatrix}$

- PolynomialFeatures are used to add complex features to the dataset.

Example: PolynomialFeatures with degree-3 transforms $\mathbf{X} = [x_1, x_2]$ into

$$\mathbf{X}' = [x_1, x_2, x_1x_2, x_1^2, x_2^2, x_1^2x_2, x_1x_2^2, x_1^3, x_2^3]$$

- KBinsDiscretizer divides a continuous variables into bins, and applies one-hot/ordinal encoding to the bin labels.

Example: KBinsDiscretizer using 5 bins transforms $\begin{bmatrix} 0 \\ 0.125 \\ 0.25 \\ 0.375 \\ 0.5 \\ 0.675 \\ 0.75 \\ 0.875 \\ 1.0 \end{bmatrix}$ into $\begin{bmatrix} 0. \\ 0. \\ 1. \\ 1. \\ 2. \\ 3. \\ 3. \\ 4. \\ 4. \end{bmatrix}$

- OneHotEncoder creates columns equal to the unique number of inputs in the column. Each column has a 1 in it and rest have 0.

Example: OneHotEncoder transforms $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}$ into $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$

- LabelEncoder encodes target labels with value from 0 and K-1, where K is the number of unique values.

1	0
2	1
6	2
1	0
8	3
6	2

Example: `LabelEncoder` transforms into

- `OrdinalEncoder` encodes categorical features with value from 0 and K-1, where K is the number of unique values. Note that `OrdinalEncoder` can operate multi dimensional data, while `LabelEncoder` can transform only 1D data.

1	'male'	0	1
2	'female'	1	0
6	'female'	2	0
1	'male'	0	1
8	'male'	3	1
6	'female'	2	0

Example: `OrdinalEncoder` transforms into

- `LabelBinarizer` creates columns equal to the unique number of inputs in the column. Each column has a 1 in it and rest have 0. Functionally, it's similar to `OneHotEncoder`, but `LabelBinarizer` is preferred on labels, while `OneHotEncoder` is preferred on features.

1	1	0	0	0
2	0	1	0	0
6	0	0	1	0
1	1	0	0	0
8	0	0	0	1
6	0	0	1	0

Example: `OrdinalEncoder` transforms into

- `MultiLabelBinarizer` creates columns equal to the unique number of inputs in the column. It's similar to `LabelBinarizer`, but for multiple labels. Thus, multiple columns can have 1 in the output.

movie_genres =		1	1	0	0
{('action', 'comedy'),		0	1	0	0
{('comedy',		1	0	0	1
{('action', 'thriller'),		1	0	1	1
{('science-fiction', 'action', 'thriller')}					

Example: `MultiLabelBinarizer` transforms to

- `add_dummy_feature` is not a transformer as such; augments the dataset with a column vector, each of whose value is 1.

7	1	1
1	8	1
2	0	1
9	6	1

Example: Using this function on

1	7	1
1	1	8
1	2	0
1	9	6

Feature Selection

- Helps at removing features that do not contribute significantly towards the model, thus leading to a decrease in dataset size and hence the computation cost.
- Filter methods:
 - `VarianceThreshold` removes all features with variance below a certain threshold. By default, the threshold is 0.
 - `SelectKBest` removes all, but the k highest-scoring features
 - `SelectPercentile` removes all, but the highest-scoring k% of features
 - `GenericUnivariateSelect` offers a generic approach for the above features, by setting *mode* to one of 'percentile', 'k_best', 'fpr', 'fdr', 'fwe'
 - `SelectFpr` selects features based on a false positive test rate.
 - `SelectFdr` selects features based on an estimated false discovery rate
 - `SelectFwe` selects features based on family-wise error rate
- Wrapper methods:

- RFE removes features recursively until the desired number of features are reached.
 - RFECV uses cross-validation to achieve the same output as RFE
 - SelectFromModel selects features based on `coef_` and `feature_importances_` from the trained estimator
 - SequentialFeatureSelector selects by either *forward selection* or *backward selection*. Note that changing the direction could typically yield different results.
- Each of the above APIs can use one of the scoring functions from
 - Mutual Information: `mutual_info_regression`, `mutual_info_classif`
NOTE: Non-negative. Higher value indicates higher dependency. 0 indicates independence.
 - Chi-square: `chi2` (can be used only for classification problems)
NOTE: Calculate between frequency of occurrence (of a feature) and label. Higher value indicates higher dependency.
 - F-statistics: `f_regression`, `f_classif`

NOTE: Mutual Information and Chi-square are recommended for sparse data.

- ColumnTransformer helps transform heterogeneous data by applying different transformers to separate subsets of features.

```
column_trans = ColumnTransformer(
    [ ('ageScaler', CountVectorizer(), [0]),
      ('genderEncoder', OneHotEncoder(dtype='int'), [1]),
      remainder='drop', verbose_feature_names_out=False)
```

Example: applies CountVectorizer on column0 and OneHotEncoder to column1 of the dataset.

- TransformedTargetRegression helps regress on some complicated function y , and return its inverse during predict.
- `sklearn.decomposition.PCA` is used to project the feature matrix or data to a lower dimensional space, by capturing bulk of the variance in as few directions as possible.
- It is important to apply exactly same transformation on training, evaluation and test set in the same order.

```
si = SimpleImputer()
X_imputed = si.fit_transform(X)
ss = StandardScaler()
X_scaled = ss.fit_transform(X_imputed)
```

Example:

- Alternatively, use Pipeline and FeatureUnion.
 - Pipeline constructs a chain of multiple transformers to execute a fixed sequence of steps in data preprocessing and modelling. It must consist a list of tuples of the form

(<step_name>, <sklearn_api/ColumnTransformer>, <list of the column numbers>).

```
estimators = [
    ('simpleImputer', SimpleImputer()),
    ('standardScaler', StandardScaler()),
]
pipe = Pipeline(steps=estimators)
pipe.fit_transform(X)
```

Example:

- FeatureUnion combines output from several Pipeline/ColumnTransformer objects by creating a new combined transformer from them.


```

num_pipeline = Pipeline([('selector', ColumnTransformer([('select_first_4',
                                                         'passthrough',
                                                         slice(0,4))])),
                        ('imputer', SimpleImputer(strategy="median")),
                        ('std_scaler', StandardScaler()),
                        ])
cat_pipeline = ColumnTransformer([('label_binarizer', LabelBinarizer(), [4]),
                                  ])
full_pipeline = FeatureUnion(transformer_list=
                              [("num_pipeline", num_pipeline),
                               ("cat_pipeline", cat_pipeline),])

```

Example:

- Note that the intermediate steps of a Pipeline must implement fit and transform, whereas the final estimator only needs to implement fit.
- Each step in the pipeline can be accessed using a 0-based index. Thus in the above example, `num_pipeline[-1]` refers to the `StandardScaler()`
- In order to access the parameters for each step, use the format `<step_name>__<parameter name>`

```

estimators = [
    ('simpleImputer', SimpleImputer()),
    ('pca', PCA()),
    ('regressor', LinearRegression())
]
pipe = Pipeline(steps=estimators)
pipe.set_params(pca__n_components = 2)

```

Example:

- Transformers can be cached by setting memory parameter in the Pipeline object.
- GridSearchCV may be applied on the pipeline in order to cross-validate with specific hyperparameters for each step. It can be used without a pipeline too, directly on the model.

Week3

- Use DummyRegressor with an appropriate strategy (mean, median, quantile, constant) to develop a baseline regressor model. Note that the strategy is based on some statistical property of the training set or a user-specified value.

SGDRegressor

- Typically used in a large training setup with greater than 10K samples. For smaller training sets, use Ridge or Lasso. The advantage of regularized models is its efficiency, which is basically linear in the number of training examples.
- Provides greater control on the optimization process through its hyperparameters
 - *loss* (squared_error, huber)
 - *penalty* (l1, l2, elasticnet)
 - *learning_rate* (constant, optimal, invscaling, adaptive). Default is *invscaling*.
 - *early_stopping* (True, False)

- If *learning_rate* is set to *invscaling*, it reduces after every iteration as $\eta^{(t)} = \frac{\eta_0}{t^{\text{power_t}}}$. Note that *eta0* and *power_t* are both hyperparameters.
- If *learning_rate* is set to *adaptive*, it is kept to the initial value as long as the training loss reduces, and gets divided by an arbitrary number (5) when the stopping criterion is reached. The algorithm stops when the learning rate goes below 10^{-6}

- Stopping criterion occurs when the training loss doesn't improve ($\text{loss} > \text{best_loss} - \text{tol}$) for $n_iter_no_change$ consecutive epochs. Alternatively, the algorithm stops when the validation score (as suggested by the parameter *scoring*) doesn't improve by at least *tol* for $n_iter_no_change$ consecutive epochs, as far as *early_stopping* is set to True and *validation_fraction* is set. By default, the algorithm stops after reaching *max_iter* in both cases.
- It's not necessary that the loss decreases from one epoch to the next, sometimes it can shoot up before reducing.
- SGDRegressor is very sensitive to feature scaling, so it is highly recommended to scale the input feature matrix, before its fit to the data.
- Training data can be shuffled after each epoch by setting parameter *shuffle* to True, during initialization.
- Converges after 10^6 training examples, so typically *max_iter* is set to $\text{np.ceil}(10^6/n)$
- When parameter *average* is set to True, the weight vector is updated to the average of weights from previous epochs. When set to an integer, the averaging starts after the set #samples.
- Setting *warm_start* to True, carries forward the weight vector from the previous run, into the current run too.

```
1 sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
2                       penalty=None, learning_rate="constant", eta0=0.0005)
3
4 for epoch in range(1000):
5     sgd_reg.fit(X_train, y_train) # continues where it left off
6     y_val_predict = sgd_reg.predict(X_val)
7     val_error = mean_squared_error(y_val, y_val_predict)
```

- All regression estimators (including SGDRegressor and LinearRegression) exposes the final set of weights learnt by the model, through attributes *coef_* and *intercept_*
- Finding weight vector can either use normal equation ($\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$), or gradient-descent technique.
- Normal equation method has a complexity of $O(m^2)$ where m is the number of features. This implies that when #features doubles, the calculation could be 4 times as slow.
- Evaluation of all models (including LinearRegression) is done using the score (or r2score) method on a object. It returns the *r2-score*, also called coefficient of determination.

residual sum of squares:

$$u = (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y})$$

Sum of squared error (actual and predicted label)

$$R^2 = \left(1 - \frac{u}{v}\right)$$

total sum of square

Sum of squared error (actual and mean predicted)

$$v = (\mathbf{y} - \hat{\mathbf{y}}_{\text{mean}})^T (\mathbf{y} - \hat{\mathbf{y}}_{\text{mean}}) \text{ where, } \hat{\mathbf{y}}_{\text{mean}} = \frac{1}{n} (\mathbf{X}\mathbf{w})$$

- The best possible *r2-score* is 1.0, when the residual sum of squares is 0
- r2-score* is 0 for a model that always predicts the same value of y , disregarding the inputs.
- The *r2-score* can be negative, because the model can be arbitrary worse, and the residual sum can be more than the total sum.
- Following is a list all possible scoring functions. Note that the scoring function are typically negative value of the errors.

Error type	Scoring	Usage
mean_absolute_error	neg_mean_absolute_error	Common
mean_squared_error	neg_mean_squared_error	Common

mean_squared_log_error	neg_mean_squared_log_error	Used when the target has exponential growths like population
root_mean_squared_error	neg_root_mean_squared_error	Common
median_absolute_error	neg_median_absolute_error	Used when outliers are present
r2	r2	-
max_error	max_error	-

- `max_error` returns maximum error due to training a model.
- In order to avoid any chances of having the **easiest** test-set that yields the least error, always perform cross-validation for robust performance evaluation.
- `sklearn` implements four cross-validation iterators:
 - `KFold` divides training data into 5 folds, and uses 4 as training and 1 for evaluation.

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import KFold
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 kfold_cv = KFold(n_splits=5, random_state=42)
7 score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

- `StratifiedKFold` works similar to `KFold`, with the exception that the folds are made preserving the percentage of samples for each class.
- `RepeatedKFold` repeats K-Fold `n` times with different randomization in each repetition.
- `LeaveOneOut`

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import LeaveOneOut
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 loocv = LeaveOneOut()
7 score = cross_val_score(lin_reg, X, y, cv=loocv)
```

- `ShuffleSplit` shuffles the order of data samples in each iteration and then splits it into train and test, and hence robust to class distribution.

```
1 from sklearn.linear_model import linear_regression
2 from sklearn.model_selection import cross_val_score
3 from sklearn.model_selection import ShuffleSplit
4
5 lin_reg = linear_regression()
6 shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
7 score = cross_val_score(lin_reg, X, y, cv=shuffle_split)
```

- In addition to the existing parameters, `cross_val_score` also accepts `scoring` parameter and can take any of the value in the table discussed under r2-score.
- To obtain test scores for each fold, use `cross_validate`. The results contain *fit_time*, *score_time*, *test_score*, *estimator*, and *train_score*

```

1 from sklearn.model_selection import cross_validate
2 from sklearn.model_selection import ShuffleSplit
3
4 cv = ShuffleSplit(n_splits=40, test_size=0.3,
5                 random_state=0)
6 cv_results = cross_validate(
7     regressor, data, target,
8     cv=cv, scoring="neg_mean_absolute_error",
9     return_train_score=True,
10    return_estimator=True)

```

NOTE: By default *estimator* and *train_score* are not returned as part of the *cross_validate* output, but only when appropriate parameters are set to True.

- *cross_validate* allows to specify multiple scoring mechanisms unlike *cross_val_score*.

Week4

- In order to use only interaction features in a polynomial transformation, set the parameter *interaction_only* to True like this. This will include only [1,x1, x2, x1x2]. x_1^2 and x_2^2 are excluded.

```

1 from sklearn.preprocessing import PolynomialFeatures
2 poly_transform = PolynomialFeatures(degree=2, interaction_only=True)

```

- Ridge Loss = Sum of squared error + regularization_rate * penalty
- Ridge regularization can be performed in two ways. Either using a *Ridge* object, or using *SGDRegressor* object with parameter *penalty* set to 'l2'
- Perform cross-validation on ridge, either using *RidgeCV* object or *GridSearchCV* on *SGDRegressor* object with parameter *penalty* set to 'l2'
- Lasso regularization can be performed in two ways. Either using a *Lasso* object, or using *SGDRegressor* object with parameter *penalty* set to 'l1'
- Perform cross-validation on lasso, either using *LassoCV* object or *GridSearchCV* on *SGDRegressor* object with parameter *penalty* set to 'l1'
- Elasticnet regularization can be performed by using *SGDRegressor* object with parameter *penalty* set to 'elasticnet' and *l1_ratio* to an appropriate value (<1)
- When *penalty* is set to 'l1' (lasso), it leads to sparse solutions.
- Penalty α is a non-negative float value. Larger values indicate stronger regularization.

List of sklearn APIs

Module	API
sklearn.feature_extraction	<ul style="list-style-type: none">- DictVectorizer- FeatureHasher
sklearn.feature_selection	<ul style="list-style-type: none">- VarianceThreshold- SelectKBest- SelectPercentile- GenericUnivariateSelect- mutual_info_regression- RFE- RFECV- SelectFromModel- SequentialFeatureSelector
sklearn.impute	<ul style="list-style-type: none">- SimpleImputer- KNNImputer- MissingIndicator
sklearn.preprocessing	<ul style="list-style-type: none">- StandardScaler- MinMaxScaler- MaxAbsScaler- FunctionTransformer- PolynomialFeatures- KBinsDiscretizer- OneHotEncoder- LabelEncoder- OrdinalEncoder- LabelBinarizer- MultiLabelBinarizer
sklearn.compose	<ul style="list-style-type: none">- ColumnTransformer- TransformedTargetRegressor
sklearn.linear_model	<ul style="list-style-type: none">- LinearRegression- SGDRegression- LogisticRegression- Lasso- LassoCV- Ridge- RidgeCV,
sklearn.dummy	DummyRegressor
sklearn.datasets	<ul style="list-style-type: none">- fetch_california_housing- make_regression

Module	API
	<ul style="list-style-type: none"> - make_blobs - make_classification - make_multilabel_classification
sklearn.model_selection	<ul style="list-style-type: none"> - train_test_split - cross_validate - cross_val_score - cross_val_predict - learning_curve - validation_curve - ShuffleSplit - permutation_test_score - GridSearchCV - RandomizedSearchCV
sklearn.metrics	<ul style="list-style-type: none"> - mean_squared_error - mean_absolute_error - mean_absolute_percentage_error
sklearn.decomposition	<ul style="list-style-type: none"> - PCA
sklearn.pipeline	<ul style="list-style-type: none"> - Pipeline - FeatureUnion
sklearn.set_config	<ul style="list-style-type: none"> - set_config
imblearn.under_sampling	<ul style="list-style-type: none"> - RandomUnderSampler
imblearn.over_sampling	<ul style="list-style-type: none"> - RandomOverSampler - SMOTE
warnings	<ul style="list-style-type: none"> - filterwarnings