

InitialDataProcessing

```
df.info()
df.shape
df.head()
df.describe()

plt.figure()
sns.countplot(x='education', hue='party', data=df, palette='RdBu')
plt.xticks([0,1], ['No', 'Yes'])
plt.show()
```

in `sns.countplot()`, we specify the x-axis data to be 'education', and hue to be 'party'. Recall that 'party' is also our target variable. So the resulting plot shows the difference in voting behavior between the two parties for the 'education' bill, with each party colored differently. We manually specified the color to be 'RdBu', as the Republican party has been traditionally associated with red, and the Democratic party with blue.

unsupervised

```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=3)
model.fit(points)
labels = model.predict(new_points)
```

Import KMeans

Create a KMeans instance with 3 clusters: model

Fit model to points

Determine the cluster labels of new_points: labels

unsupervised (cont)

```
centroids = model.cluster_centers_
df = pd.DataFrame({'NameOfArray1': array1, 'NameOfArray2': array2})
pd.crosstab([df['NameOfArray1'], df['NameOfArray2']])
```

Assign the cluster centers: `centroids`. note that model was KMeans- (`n_clusters=k`)

Create a DataFrame with arrays as columns: `df`

It is a table where it contains the counts the number of times each array2 coincides with each array1 label.

Classification

```
X = df.drop('targetvariable', axis=1).values
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X, y)
```

Note the use of `.drop()` to drop the target variable from the feature array X as well as the use of the `.values` attribute to ensure X are NumPy arrays

Instantiate a `KNeighborsClassifier` called `knn` with 6 neighbors by specifying the `n_neighbors` parameter.

the classifier to the data using the `.fit()` method. X is the features, y is the target variable

```
from sklearn.neighbors import KNeighborsClassifier
```

Import `KNeighborsClassifier` from `sklearn.neighbors`

```
knn.predict(X_new)
```

Predict for the new data point `X_new`

Classification (cont)

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, random_state=42, stratify=y)

knn.score(X_test, y_test)

np.arange(1, 9)

for counter, value in enumerate(some_list):
    print(counter, value)
```

from `sklearn.model_selection` import `train_test_split`

Create stratified training and test sets using 0.2 for the size of the test set. Use a random state of 42. Stratify the split according to the labels so that they are distributed in the training and test sets as they are in the original dataset.

Compute and print the accuracy of the classifier's predictions using the `.score()` method.

numpy array from 0 to 8=`np.arange(1, 9)`

`Enumerate` is a built-in function of Python. It's usefulness can not be summarized in a single line. Yet most of the newcomers and even some advanced programmers are unaware of it. It allows us to loop over something and have an automatic counter.

Classification (cont)

```
my_list = ['apple', 'banana', 'grapes', 'pear']
for c, value in enumerate(my_list, 1):
    print(c, value)
```

Output: # 1
apple # 2
banana # 3
grapes # 4
pear

Regression

```
df['Co-Name1'].corr(df['ColName2'])
```

Caluclate the correlation between ColName1 and ColName2 in dataframe df

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

Returns number spaces evenly w.r.t interval. Similiar to arange but instead of step it uses sample number. Parameters : -> start : [optional] start of interval range. By default start = 0 -> stop : end of interval range -> restep : If True, return (samples, step). By deflut restep = False -> num : [int, optional] No. of samples to generate -> dtype : type of output array

```
from sklearn.linear_model import LinearRegression
```

Import LinearRegression

```
from sklearn.metrics import mean_squared_error
```

```
from sklearn.metrics import mean_squared_error
```

Regression (cont)

```
mean_squared_error(y_true, y_pred, sample_weight=None, multioutput='uniform_average')
```

Mean squared error regression loss

```
from sklearn.model_selection import cross_val_score
```

```
reg = LinearRegression()
```

Create a linear regression object: reg

```
cv_scores = cross_val_score(reg, X, y, cv=5)
```

Compute 5-fold cross-validation scores: cv_scores

```
from sklearn.linear_model import Lasso
```

Import Lasso

```
lasso = Lasso(alpha=0.4, normalize=True)
```

Instantiate a lasso regressor: lasso

```
lasso.fit(X, y)
```

Fit the regressor to the data

```
lasso_coef = lasso.coef_
```

Compute and print the coefficients

```
from sklearn.linear_model import Ridge
```

Import necessary modules

Regression (cont)

```
def display_plot(cv_scores, cv_scores_std):
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.plot(alpha_space, cv_scores)
    std_error = cv_scores_std / np.sqrt(10)
    ax.fill_between(alpha_space, cv_scores - std_error, cv_scores + std_error, alpha=0.2)
    ax.set_ylabel('CV Score +/- Std Error')
    ax.set_xlabel('-Alpha-')
    ax.axhline(np.max(cv_scores), linestyle='--', color='.5')
    ax.set_xlim([alpha_space[0], alpha_space[-1]])
    ax.set_xscale('log')
    plt.show()
```

you will practice fitting ridge regression models over a range of different alphas, and plot cross-validated R2 scores for each, using this function that we have defined for you, which plots the R2 score as well as standard error for each alpha:

```
cross_val_score(Ridge(normalize=True), X, y, cv=10)
```

perform 10-fold CV for Rdige Regressin.