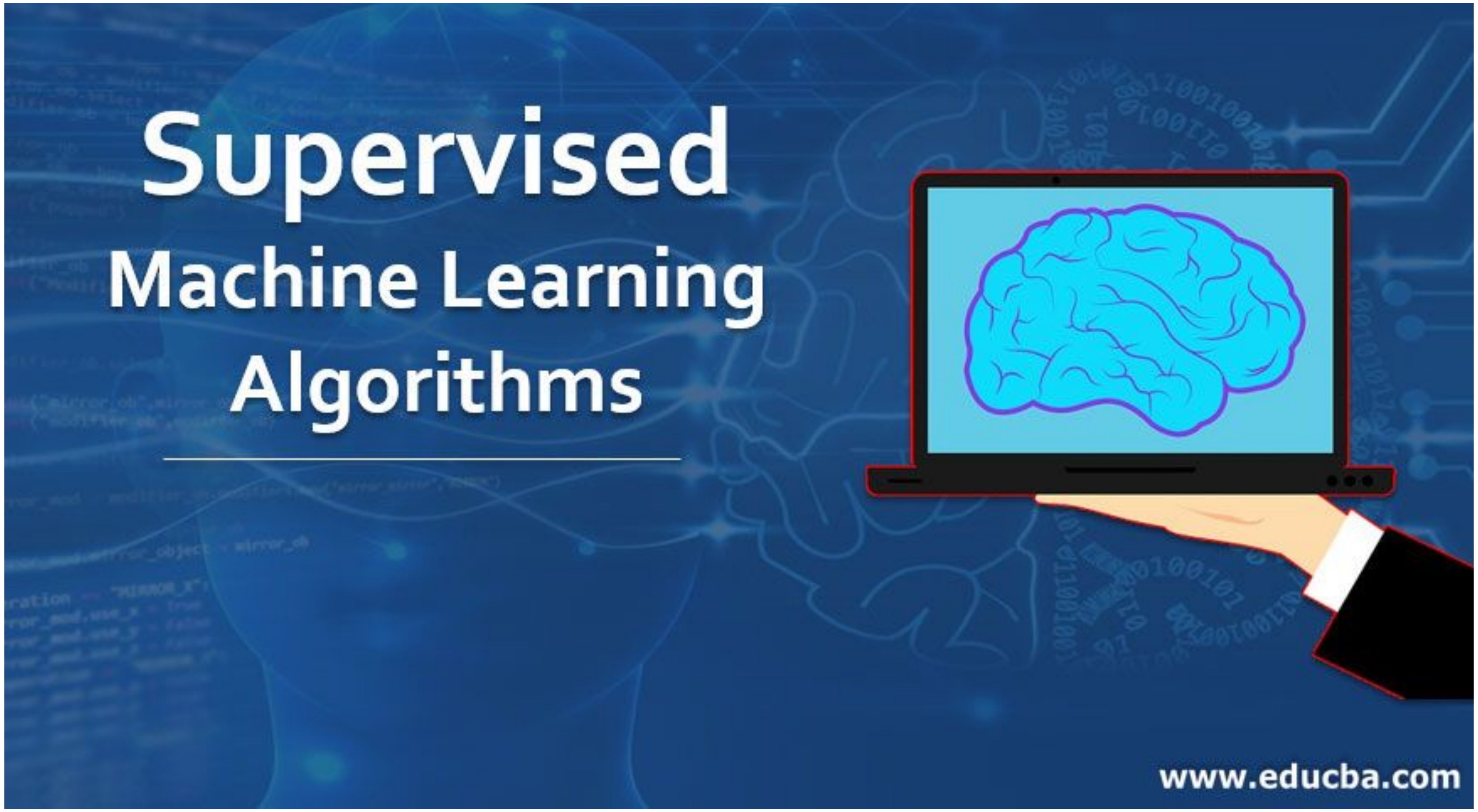


ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

Last lecture reminder

We learned about:

- Introduction to Polynomial Regression
- Polynomial Regression - Feature interaction
- Model overfitting and model underfitting
- The Bias-Variance Tradeoff
- Find Most Fit Polynomial Degree
- Polynomial Regression - Model deployment and import

Feature Scaling

Feature Scaling → Feature scaling is a preprocessing technique used in machine learning to standardize or normalize the range of independent variables or features of data. This is important because features of different scales can skew the learning process in favor of the features with larger magnitudes. Feature scaling makes sure that all features contribute equally to the model's learning process.

Pros of feature scaling:

- **Avoids Dominance of Features** → In some algorithms If one of the features has a broad range of values, the model measurement will be dominated by this feature, making the model unable to learn from other features correctly.
- **Helps in Learning Phase** → Algorithms like Neural Networks, that involve weight inputs, will not have an optimal, speedy learning phase if not scaled.
- **Improves model's performance** → Scaled features can improve the model's performance, especially when the dataset contains features of different scales.

Feature Scaling

Cons for feature scaling:

- **Complicate the model data preparation** → We will always need to repeat the scaling process on new data in order to use our algorithm.
- **Hard to revert to original units** → Once we decided to apply feature scaling, the prediction result will be in scaled units and not the original. In most cases it will be hard for us to take those results and convert them back to the original units.
- **Information Loss** → If not done correctly, scaling can lead to loss of information. For instance, outliers may be lost when we scale features using a method that is sensitive to outliers.

Note: Not all algorithms require feature scaling. Decision trees and Random Forest algorithms are not affected by feature scaling. On the other hand supervised learning algorithms based on Gradient Descent, Distance and K-Means Clustering will required feature scaling for better prediction results.

Feature Scaling Methods

There are two common methods of feature scaling:

Normalization → This method scales the features such that they have a value between 0 to 1. It calculates this by subtracting the minimum value of the feature and then dividing by the range of that feature.

$$X_{changed} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

- X - represents a value of the feature.

- Xmin - represents the smallest value of the feature.

- Xmax - represents the biggest value of the feature.

- Xchange - represents the normalized value of X.

ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

Feature Scaling Methods

Standardization → This method scales the features such that they have a mean of 0 and a standard deviation of 1. This involves subtracting the mean value of the feature and then dividing by its standard deviation.

$$X_{changed} = \frac{X - \mu}{\sigma}$$

- X represents a value of the feature.

- μ(mu) - represents the mean (average) of the feature.

- σ(sigma) - represents the standard deviation of the feature.

- Xchange - represents the standardized value of X.

Feature Scaling Process

When applying feature scaling we have to execute the following steps:

- Perform train / test split
- Fit to training feature data
- Transform training feature data
- Transform test feature data

Fit training feature data → Fit mean calculate the parameters needed for the feature scaling, for example the mean and sd of the data. Here <u>it's important to remember not to include in your calculations the test data set</u> (the mean and sd should be calculated only from the train set).
This is because <u>the test data set should be unknown data so it should not affect the scaling process of the known data</u> (the train data set).
When we expose the model to data that should be unseen (like the test data set) we are causing something that called **Data Leakage**

Introduction To Regularization

Regularization → A technique used in supervised learning that deals with the issue of overfitting, where a model learns the training data too well, to the point that it performs poorly on unseen data (or test data). Overfitting typically results in a model that is too complex with excessive parameters or coefficients.

Regularization seeks to address overfitting by adding a penalty term to the loss function, which discourages the learning algorithm from assigning too much importance to any individual feature, thus reducing the complexity of the model. It essentially imposes a cost on complexity.

Note: When applying model regularization, feature scaling is a must this is because we want to make sure all features are on a similar scale. This way, the regularization term will penalize all coefficients in the same manner, ensuring a fair shrinkage.

Introduction To Regularization

When should we use regularization?

- **Model Overfitting** → If your model is overfitting the data, meaning it performs well on the training data but poorly on the validation/test data, this is the key situation where regularization is helpful.
- **High Variance of Model Predictions** → If your model has high variance, which means it's sensitive to small fluctuations in the input data, it may be a good candidate for regularization.
- **Large Number of Features** → When dealing with datasets with a large number of features, or when features are highly correlated, regularization can help by effectively reducing the number of features or managing multicollinearity.
- **You Want to Keep the Model Simple** → Regularization can help achieve a simpler model by adding a complexity penalty. Besides combating overfitting, simpler models are often easier to interpret.

L2 Regularization - Ridge Regression

L2 Regularization (also known as Ridge regularization) → This regression type adds a term equal to the square of the magnitude of coefficients to the loss function. L2 doesn't exclude parameters completely but makes their contribution smaller by reducing their values.

In order to understand what is the penalty term in ridge regression we need to perform the following calculation:

As we learned, in linear regression what we want is to minimize the cost function which build from the difference between the observation value and the prediction value. We called this difference residual.

We can call this cost function RSS - residual sum of squares.

$$\mathrm{RSS} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

L2 Regularization - Ridge Regression

We can now take that RSS formula and replace ŷ with the linear regression equation:

$$\hat{y} = \beta_0 x_0 + \cdots + \beta_n x_n$$

$$\begin{aligned}\mathrm{RSS} &= \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\ &= \sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_{i1} - \hat{\beta}_2 x_{i2} - \cdots - \hat{\beta}_p x_{ip})^2\end{aligned}$$

So for p different features we can summarize RSS as:

$$\mathrm{RSS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2$$

ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

L2 Regularization - Ridge Regression

Ridge regression adds a shrinkage penalty, meaning when apply ridge regression our cost function will contain the same RSS as with regular linear regression but with additional value that penalizes the size of the coefficients. Our model will now need to, not only minimize the error, but also keep the feature coefficients as small as possible.

The new cost function that our model will try to minimize will look like this:

$$\text{Error} = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 \boxed{+ \lambda \sum_{j=1}^{p}\beta_j^2}$$

The addition term takes the square of every feature coefficient and multiply it by λ (lamda)

The λ (lamda) can be any value that we want from 0 (result regular RSS) to infinity. The higher the lambda value the more severe the penalty

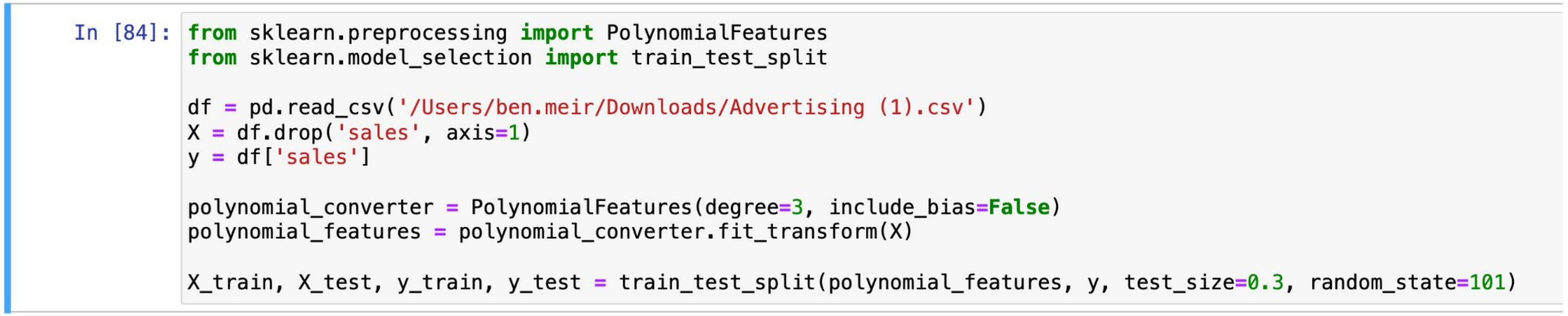
ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

Ridge Regression - Python Example

Let’s now see a real Python example appling ridge regression model.

For this example we will use the **‘Advertising.csv’** file and because we are going to add additional penalty value to our cost function, we need to make sure our data is in similar units (feature scaling).

First, let’s create a third-degree polynomial model so our model will be complex enough, what that can result an overfitting.



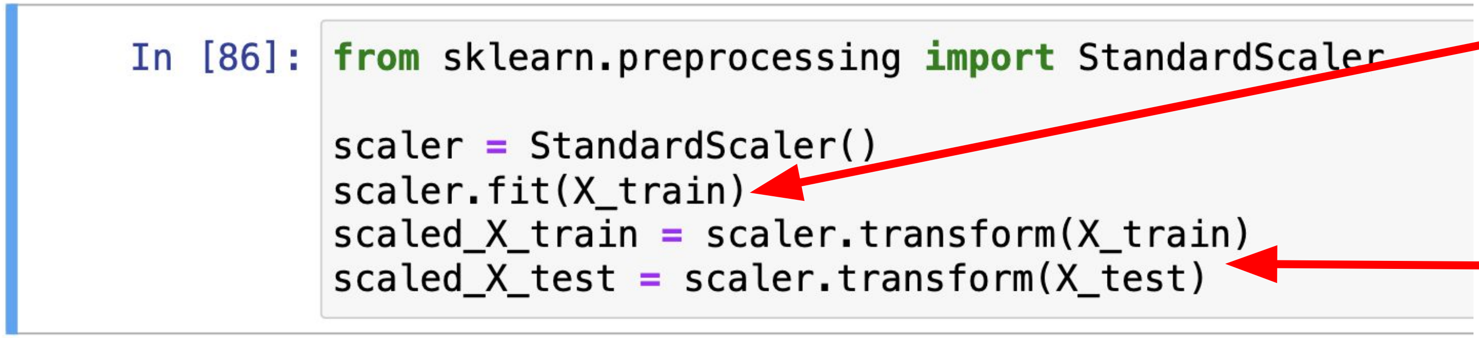
```
In [84]: from sklearn.preprocessing import PolynomialFeatures
         from sklearn.model_selection import train_test_split

         df = pd.read_csv('/Users/ben.meir/Downloads/Advertising (1).csv')
         X = df.drop('sales', axis=1)
         y = df['sales']

         polynomial_converter = PolynomialFeatures(degree=3, include_bias=False)
         polynomial_features = polynomial_converter.fit_transform(X)

         X_train, X_test, y_train, y_test = train_test_split(polynomial_features, y, test_size=0.3, random_state=101)
```

ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

Ridge Regression - Python Example

Now, because we want to use Ridge Regression model we first need to apply feature scaling.

For this example we will use the standardization type feature scaling.

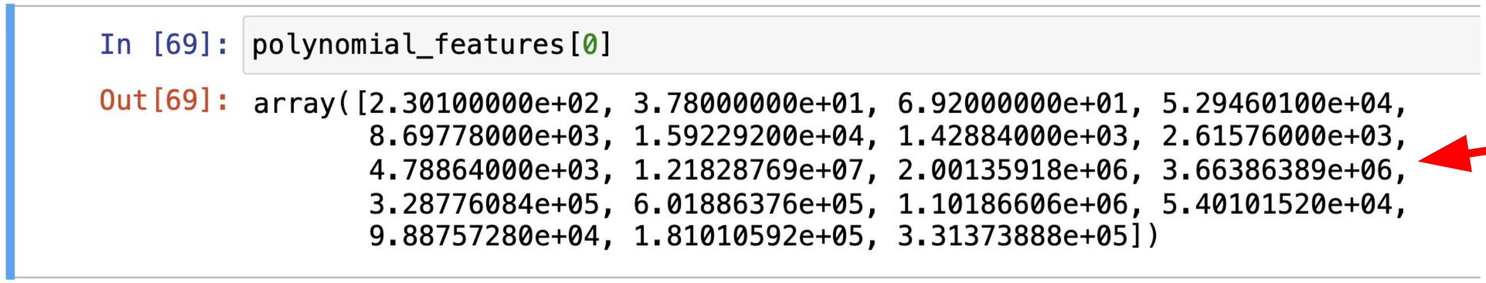


```
In [86]: from sklearn.preprocessing import StandardScaler

         scaler = StandardScaler()
         scaler.fit(X_train)
         scaled_X_train = scaler.transform(X_train)
         scaled_X_test = scaler.transform(X_test)
```

We are fitting our scaler only to the train data set so we won't cause data leakage

Once our scaler calculate what it need for the feature scaling we can apply the transformation on our train data set and our test data set

Let's take a look on the scaled results and see if we actually managed to reduce the size of our feature values to the same size units:

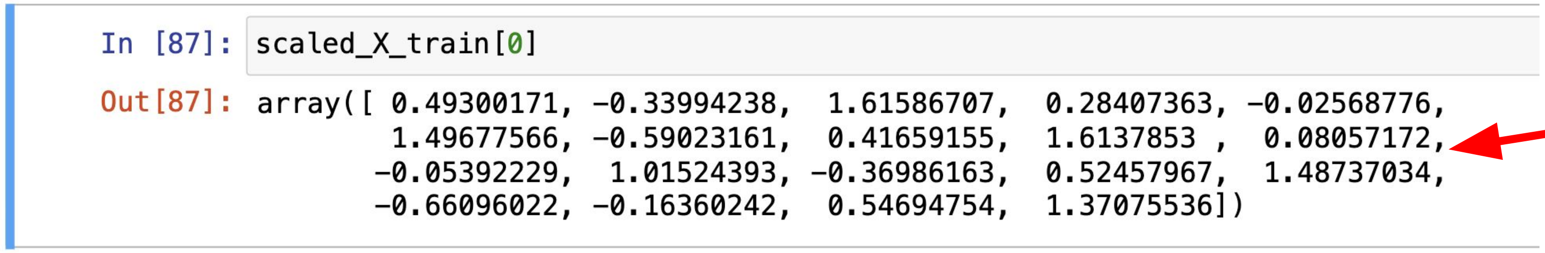


```
In [69]: polynomial_features[0]

Out[69]: array([2.30100000e+02, 3.78000000e+01, 6.92000000e+01, 5.29460100e+04,
                8.69778000e+03, 1.59229200e+04, 1.42884000e+03, 2.61576000e+03,
                4.78864000e+03, 1.21828769e+07, 2.00135918e+06, 3.66386389e+06,
                3.28776084e+05, 6.01886376e+05, 1.10186606e+06, 5.40101520e+04,
                9.88757280e+04, 1.81010592e+05, 3.31373888e+05])
```

The original data contain very large feature values (because the high degree we chose)

Ridge Regression - Python Example

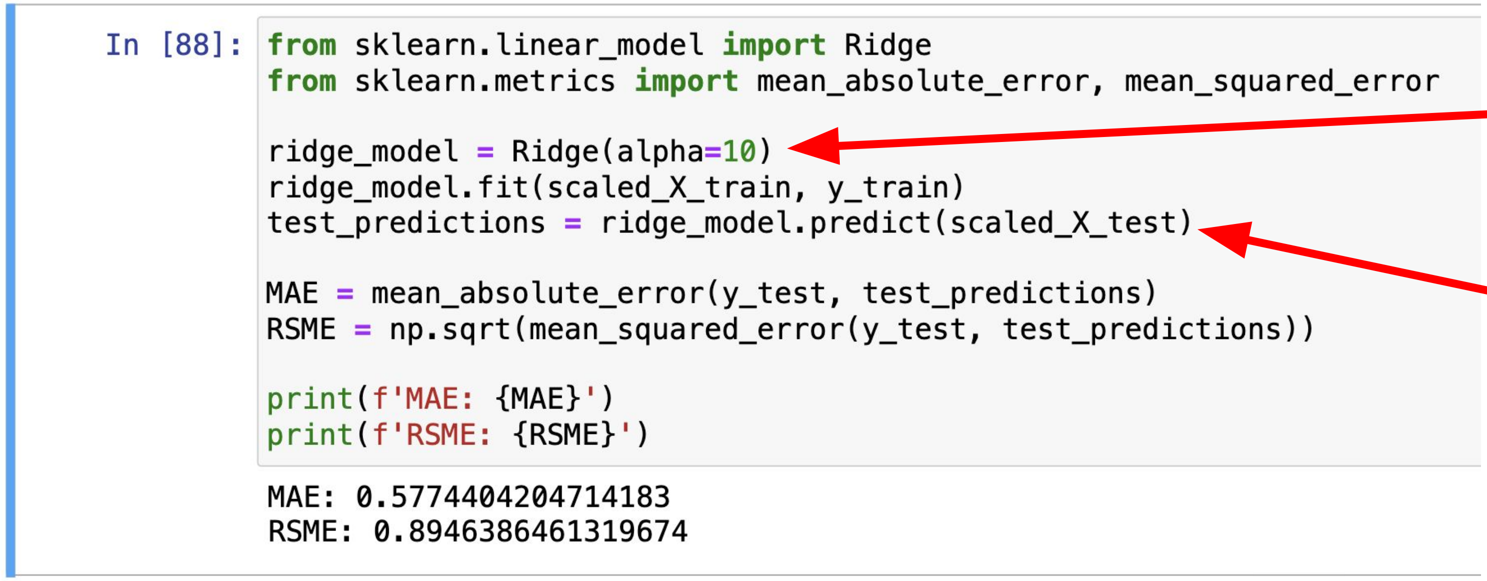
Now, let's see our scaled data:



```
In [87]: scaled_X_train[0]

Out[87]: array([ 0.49300171, -0.33994238,  1.61586707,  0.28407363, -0.02568776,
                 1.49677566, -0.59023161,  0.41659155,  1.6137853 ,  0.08057172,
                -0.05392229,  1.01524393, -0.36986163,  0.52457967,  1.48737034,
                -0.66096022, -0.16360242,  0.54694754,  1.37075536])
```

We can see that our scaled data for the first row indeed contain much smaller values

Now, let's perform Ridge Regression on our scaled data set and print the MAE & RSME results:



```
In [88]: from sklearn.linear_model import Ridge
         from sklearn.metrics import mean_absolute_error, mean_squared_error

         ridge_model = Ridge(alpha=10)
         ridge_model.fit(scaled_X_train, y_train)
         test_predictions = ridge_model.predict(scaled_X_test)

         MAE = mean_absolute_error(y_test, test_predictions)
         RSME = np.sqrt(mean_squared_error(y_test, test_predictions))

         print(f'MAE: {MAE}')
         print(f'RSME: {RSME}')

         MAE: 0.5774404204714183
         RSME: 0.8946386461319674
```

We chose the alpha (which is the same as the lambda value in our formula) to be 10

We are using the scaled data feature values and not the original values

Ridge Regression - Lambda Value

In the previous example we chose ‘10’ to be our lambda (alpha in the model) value. We know that choosing the right lambda value is important because it determine the size of penalty our model will get for each beta coefficient.

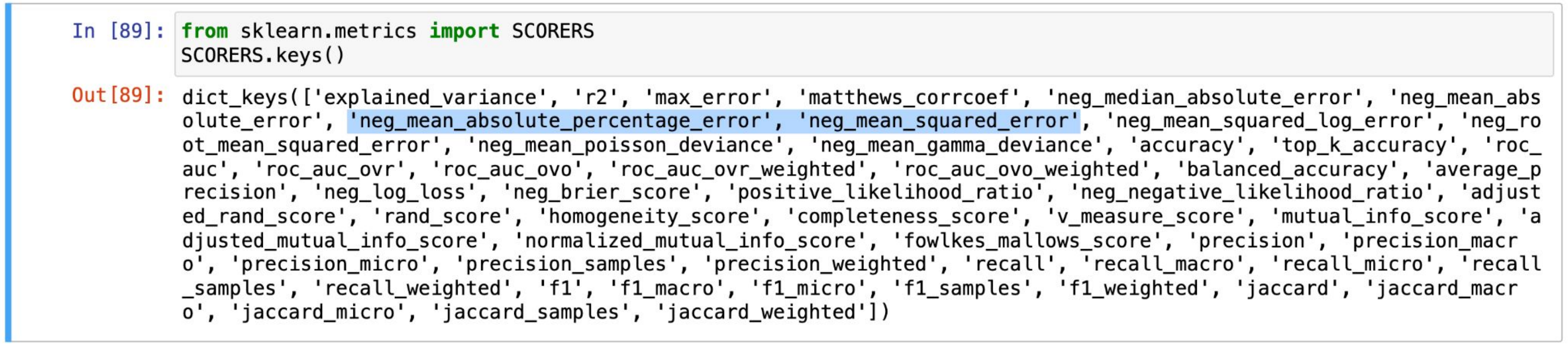
How can we know what is the right lambda value?
Fortunately Scikit-Learn comes with model called **RidgeCV** (CV - Cross Validation) which basically run multiple Ridge models with different lambda values and choose the lambda that return the best results. In order to use it we need to provide the RidgeCV model the range of lambda values and the scoring metric.

Note: Cross-validation can significantly increase computational load, because the training and validation process is being carried out multiple times on different subsets of the data.

Ridge Regression - Lambda Value

Scoring metric → Scikit-Learn will determine what is the optimal lambda value according to the scoring metric we provide. The logic it based on is the larger the score the better the result.
In our case, it's not true because we want to find the lambda that minimize the error so the lower the error the better the lambda value. In order to keep the Scikit-Learn scoring logic what we want is to multiply the error with -1 so it will indeed be the higher the score the better the result.
This is why we will pass scoring like - ‘neg_mean_absolute_error’ or ‘neg_root_mean_squared_error’.

Here we can see all the scoring metrics options Scikit-Learn provide:

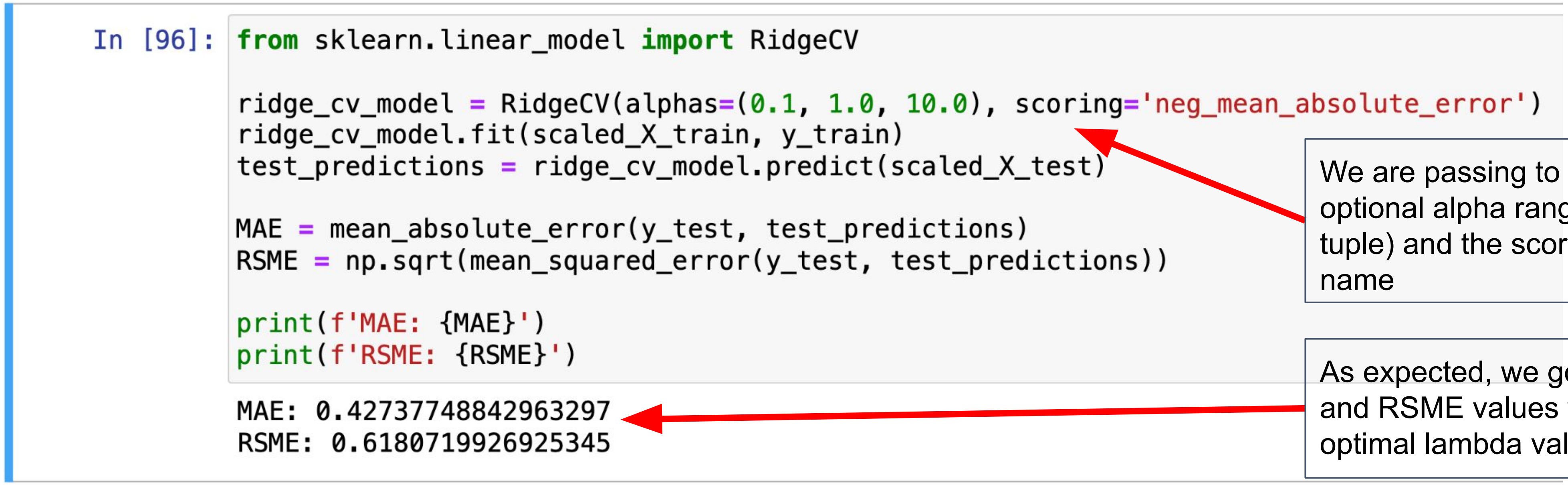


```
In [89]: from sklearn.metrics import SCORERS
         SCORERS.keys()
```

```
Out[89]: dict_keys(['explained_variance', 'r2', 'max_error', 'matthews_corrcoef', 'neg_median_absolute_error', 'neg_mean_abs
         olute_error', 'neg_mean_absolute_percentage_error', 'neg_mean_squared_error', 'neg_mean_squared_log_error', 'neg_ro
         ot_mean_squared_error', 'neg_mean_poisson_deviance', 'neg_mean_gamma_deviance', 'accuracy', 'top_k_accuracy', 'roc_
         auc', 'roc_auc_ovr', 'roc_auc_ovo', 'roc_auc_ovr_weighted', 'roc_auc_ovo_weighted', 'balanced_accuracy', 'average_p
         recision', 'neg_log_loss', 'neg_brier_score', 'positive_likelihood_ratio', 'neg_negative_likelihood_ratio', 'adjust
         ed_rand_score', 'rand_score', 'homogeneity_score', 'completeness_score', 'v_measure_score', 'mutual_info_score', 'a
         djusted_mutual_info_score', 'normalized_mutual_info_score', 'fowlkes_mallows_score', 'precision', 'precision_macr
         o', 'precision_micro', 'precision_samples', 'precision_weighted', 'recall', 'recall_macro', 'recall_micro', 'recall
         _samples', 'recall_weighted', 'f1', 'f1_macro', 'f1_micro', 'f1_samples', 'f1_weighted', 'jaccard', 'jaccard_macr
         o', 'jaccard_micro', 'jaccard_samples', 'jaccard_weighted'])
```

RidgeCV - Python Example

Let's now perform RidgeCV model and let Scikit-Learn choose the optimal lambda value for us:

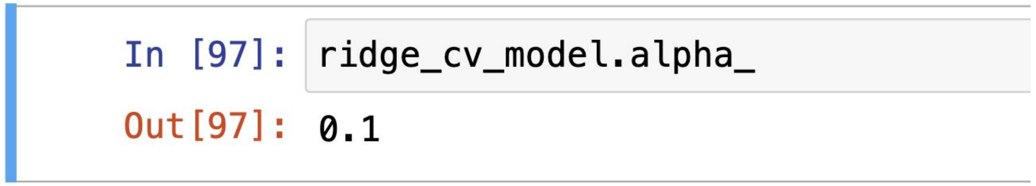


```
In [96]: from sklearn.linear_model import RidgeCV

         ridge_cv_model = RidgeCV(alphas=(0.1, 1.0, 10.0), scoring='neg_mean_absolute_error')
         ridge_cv_model.fit(scaled_X_train, y_train)
         test_predictions = ridge_cv_model.predict(scaled_X_test)

         MAE = mean_absolute_error(y_test, test_predictions)
         RSME = np.sqrt(mean_squared_error(y_test, test_predictions))

         print(f'MAE: {MAE}')
         print(f'RSME: {RSME}')

         MAE: 0.42737748842963297
         RSME: 0.6180719926925345
```

We are passing to RidgeCV() the optional alpha range values (as a tuple) and the scoring metric name

As expected, we got better MAE and RSME values when using the optimal lambda value

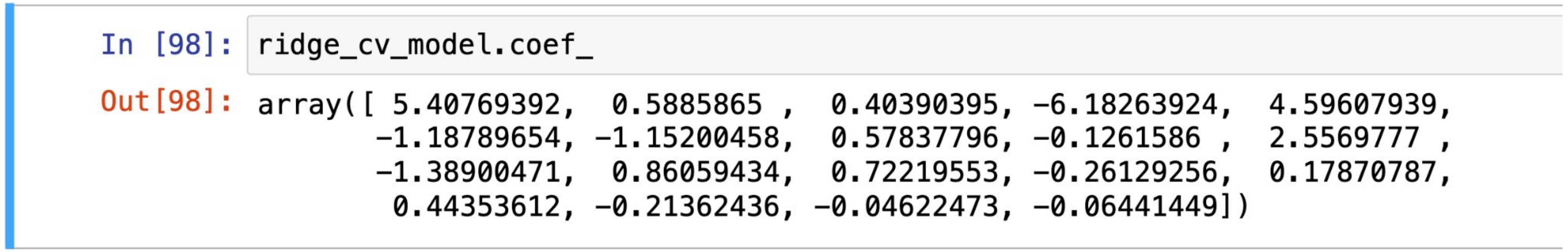
If we want to see what was the optimal lambda value that been used in our Ridge model we can execute:



```
In [97]: ridge_cv_model.alpha_

Out[97]: 0.1
```

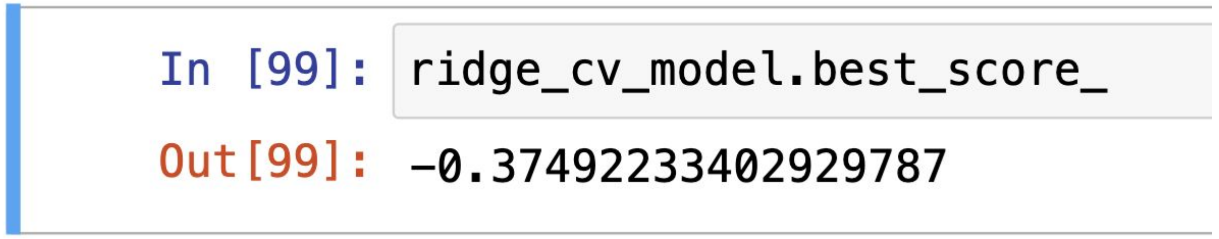
RidgeCV - Python Example

We can also take a look on the different coefficients our RidgeCV model gave to each feature:



```
In [98]: ridge_cv_model.coef_

Out[98]: array([ 5.40769392,  0.5885865 ,  0.40390395, -6.18263924,  4.59607939,
                -1.18789654, -1.15200458,  0.57837796, -0.1261586 ,  2.5569777 ,
                -1.38900471,  0.86059434,  0.72219553, -0.26129256,  0.17870787,
                 0.44353612, -0.21362436, -0.04622473, -0.06441449])
```

And we can also see what was the best score for the optimal lambda value (the lambda value that produced the highest negative mean absolute error):



```
In [99]: ridge_cv_model.best_score_

Out[99]: -0.37492233402929787
```

Class Exercise - Ridge Regression

Instructions:

Use the data set provided in the previous class exercise and use **polynomial regression & Ridge** model training.

Hours of calling customers: [2, 3, 4, 5, 6, 1.5, 5, 7, 8, 10]

Money earned: [50K$, 70K$, 90K$, 100K$, 110K$, 40K$, 110K$, 130K$, 145K$, 180K$]

- Convert your model data to be with polynomial regression of 4 degree
- Use feature scaling of type Normalization to prepare your data set
- Use basic Ridge model to eliminate overfitting and choose alpha value of 5
- Print the MAE and RMSE results of your model
- Use RidgeCV and find what is the optimal alpha value for range between 1 to 10 with jumps of 0.1 and use ‘neg_root_mean_squared_error’ as the score metric

Class Exercise - Ridge Regression

Instructions:

- Print the MAE and RMSE results of your RidgeCV model and see if your results indeed improved your model accuracy
- Print the beta coefficients your model found for each feature
- Print the optimal alpha and the score value that this alpha got

Class Exercise Solution - Ridge Regression

ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק