Data modelling with sikit - learn

When creating models, we need to figure out the optimal hyperparameters. Hyper parameters are values that we set when creating a model.

Linear regression

One of the main objectives in both ML and DS is finding an equation or distribution that best fits a given dataset. This is known as data modelling.

The term linear regression refers to using a linear model to represent the relationship between a set of independent variables and a dependent variable

$$y = ax_1 + bx_2 + cx_3 + d$$

The simplest form of linear regression is called least squares regression.

The fit function takes in an input dataset of features (independent variables) and an array of labels (dependent variables) for each data observation (rows of the dataset)

```
from sklearn import linear_model
reg = linear_model.LinearRegression()
reg.fit(pizza_data, pizza_prices)
```

Scanned by CamScanner

After calling the fit function, the model is ready to use. The predict function allows us to make predictions.

We can also get specific coefficients and intercept for the linear combination using the coef_ and intercept_ properties.

We can retrieve the coefficient of determination R^2 using the score function

```
price_predicted = reg.predict(new_pizzas)
print(reg.coef_)
print(reg.intercept_)

r2 = reg.score(pizza_data, pizza_prices)
```

The R^2 value tells us how close of a fit the linear model is to the data or in other words, how good of a fit the model is for the data.

The traditional R^2 value is a real number between 0 and 1. In scikit learn, it ranges from $-\infty$ to 1, where lower values denote a poorer fit to the data. The closer the value is to 1, the better the model's fit on the data.

Scanned by CamScanner

Ridge Regression

Least square regression relies on the fact that the dataset's features are each independent ie uncorrelated. When this is not the case, it makes the least squares regression model highly sensitive to noise in the data

Because real life data tends to have noise, and will often have some linearly correlated features in the dataset, we combat this by performing regularization.

For ordinary least square regression, the goal is to find the weights for the linear model that minimise the sum of squared residuals:

$$\sum_{i=1}^{n} (x_i w - y_i)^2$$

Where x_i represents a data observation and y_i is the label.

The most commonly used form of regularisation is ridge regularisation.
With ridge regularisation, the goal is to now find the weights that minimise the following quantity

$$\alpha \|w\|_2^2 + \sum_{i=1}^{n} (x_i w - y_i)^2$$

Where α is a non negative real number hyperparameter

Scanned by CamScanner

and $\|w\|_2$ represents the L2 norm of the weights. The additional $\alpha\|w\|_2^2$ is referred to as the <u>penalty term</u> as it penalises larger weight values

→
```
from sklearn import linear_model
reg = linear_model.Ridge(alpha=0.1)
reg.fit(pizza_data, pizza_prices)
r2 = reg.score(pizza_data, pizza_prices)
```

We can specify the value of a hyperparameter when initialising the Ridge object (default is 1.0) However, rather than manually choosing a value, we can use cross-validation to help us choose the optimal α from a list

→
```
from sklearn import linear_model
alphas = [0.1, 0.2, 0.3]
reg = linear_model.RidgeCV(alphas=alphas)
reg.fit(pizza_data, pizza_prices)
```

Scanned by CamScanner

Lasso regression.

LASSO uses L1 norm for the weights penalty term
LASSO regularisation will find optimal weights to minimise

$$\alpha \|w\|_1 + \sum_{i=1}^{n} (x_i w - y_i)^2$$

where $\|w\|_1$ represents the L1 norm of the weights

LASSO regularization tends to prefer linear models with fewer parameter values. This means that it will likely zero 'out some of the weight coeff.s. This reduces the number of features that the model is actually dependent on.

→
```
from sklearn import linear_model
reg = linear_model.Lasso(alpha=0.1)
reg.fit(data, labels)
reg.score(data, labels)
```

Scanned by CamScanner

Bayesian Regression.

The main idea is to make a certain assumptions about the probability distribution of a model's parameters before being fitted on the data. These initial distribution assumptions are called priors

There are two hyperparameters to optimise: α and λ
The λ hyperparameter acts as the precision of the model's weights. Smaller the λ values greater the variance between individual weight values.

Both α and λ have gamma distribution priors, meaning we assume both values come from a gamma probability distribution.

α parameter has the prior : $\Gamma(\alpha_1, \alpha_2)$
λ " " " " : $\Gamma(\lambda_1, \lambda_2)$
where $\Gamma(k, \theta)$ represent a gamma distribution with shape parameter k and scale param θ

→
```
from sklearn import linear_model
reg = linear_model.BayesianRidge()
reg.fit(data, labell)
```

we can manually specify the α_1 and α_2 gamma params with alpha_1 and alpha_2 when initialising BayesianRidge(). Similarly we can also manually set lamda_1 and lambda_2.
Default value for all is 10^{-6}

Scanned by CamScanner

Logistic Regression

It is a linear model for classification. It is called logistic regression because it performs regression on logits, which then allows us to classify the data based on model probability predictions

```
from sklearn import linear_model
reg = linear_model.LogisticRegression()
reg.fit(data, labels)
reg.predict(classes)
```

For multiclass classification, we initialise the LogisticRegression object with multi_class argument. The default value is 'ovr' which signifies a one-vs-rest strategy. In multiclass, we use 'multinomial'.

To use 'multinomial' strategy, we need to choose a proper solver. In this case we use 'lbfgs'

```
from sklearn import linear_model
reg = linear_model.LogisticRegression(
        solver='lbfgs',
        multi_class='multinomial')
reg.fit(data, labels)
reg.predict(new_data)
```

Logistic Regression object uses a solver to obtain the optimal weight settings based on the input data and labels

Scanned by CamScanner

→ from sklearn import linear_model
reg = linear_model.LogisticRegression(
solver='lbfgs', max_iter=1000)

Cross validated model.
→ from sklearn import linear_model
reg = linear_model.LogisticRegressionCV(
solver='multinomial',
max_iter = 1000
)

Scanned by CamScanner

Decision trees

It is a binary tree where each node of the tree decides on a particular feature of the dataset and we descend to the node's left or right child depending on the feature's value.

The leaves of the decision tree determine the class label to predict (in classification) or the real number value to predict (in regression)

```
from sklearn import tree
clf_tree1 = tree.DecisionTreeClassifier()
reg_tree1 = tree.DecisionTreeRegressor()
clf_tree2 = tree.DecisionTreeClassifier(
            max_depth=8)
clf_tree1.fit(data, labels)
```

The max_depth keyword argument lets us manually set the maximum number of layers allowed in the decision tree. Default value is None meaning that the decision tree will continue to be constructed until no nodes can have children anymore.

Since large decision trees are prone to overfit data, it can be benificial to manually set a maximum depth for the tree.

The question is how we choose features to

Scanned by CamScanner

decide on at each node. In general terms, we want to choose the feature value that best splits the remaining dataset at each node

Since scikit-learn uses the CART algorithm, we use Gini Impurity, MSE (mean squared error) and MAE (mean absolute error) to decide on the best feature at each node. The default for DecisionTreeRegressor is MSE.

Scanned by CamScanner

Training and testing.

```
from sklearn.model_selection import train_test_split
split_dataset = train_test_split (data, labels)
train_data = split_dataset [0]
test_data =      "          [1]
train_labels =   "          [2]
test_labels =    "          [3]
```

train_test_split function randomly shuffles the dataset and corresponding labels prior to splitting. This is good practice to remove any systematic orderings in the dataset, which could potentially impact the model.

The default size of testing set is 25%. We can use test_size keyword argument to manually specify the proportion of the original dataset that will go into the testing set

```
→ split_dataset = train_test_split (
                    data, labels, test_size = 0.375)
```

Scanned by CamScanner

Cross validation.

If the dataset is big enough, we split it into 3 - training, testing and validation. The validation set is about the same size as the testing set and it is used for evaluating the model after training. The testing set is then used for final evaluation.

However, partitioning the data set into three will not work if the dataset is small. A solution to this problem is cross validation.

Cross validation creates synthetic validation sets by partitioning the training set into multiple smaller subsets. K-fold CV partitions the training set into K approximately equal sized subsets referred to as "folds". There are K rounds of the algorithm and each round chooses one of the K subsets for the validation sets

```
from sklearn import linear_model
from sklearn.model_selection import cross_val_score
clf = linear_model.LogisticRegression()
cv_score = cross_val_score(clf, data, labels, cv=3)
```

Note - we dont call fit with the model prior to using cross_val_score.

Scanned by CamScanner

For decision trees, we can tune the tree's max depth hyperparameter by using K-fold.

```
is_clf = True      # for classification
for depth in range(3, 8):
    scores = cv_decision_tree(
                is_clf, data, labels, depth, 5) #K=5
    mean = scores.mean()      # mean accross all folds
    std_2 = 2* scores.std()   # 2 std devs
```

Evaluation

For classification, we use classification accuracy.
For regression, we normally use either R^2 values, mean squared error or mean absolute error

```
# regression model's predictions
reg = tree.DecisionTreeRegressor()
reg.fit(data, labels)
predictions = reg.predict(test_data)

from sklearn import metrics
r2 = metrics.r2_score(test_labels, predictions)
mse = metrics.mean_squared_error(" , " )
mae = metrics.mean_absolute_error(" , " )

# for classification
clf = tree.DecisionTreeClassifier()
clf.fit(data, labels)
predictions = clf.predict(test_data)
from sklearn import metrics
acc = metrics.accuracy_score(test_labels, predictions)
```

Scanned by CamScanner

Exhaustive tuning

If the dataset is small, we can apply an exhaustive grid search for tuning hyperparams. We specify each possible value for each hyperparameter and then search will go through each possible combination and return the model with the best combination.

→ reg = linear_model.BayesianRidge()
params = {
'alpha_1': [0.1, 0.2, 0.3],
'alpha_2': [0.1, 0.2, 0.3]
}

reg_cv = GridSearchCV(reg, params, cv = 5, iid = False)
reg_cv.fit(train_data, train_labels)
print(reg_cv.best_params_)

cv → number of folds used in K-folds
iid → how cv score is calculated

Scanned by CamScanner