ML Tutorial 3

June 13, 2018

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import sklearn
        import seaborn as sns
        sns.set()
        %matplotlib inline
```

0.1 Modelling in Python

0.1.1 Setting up and Encoding

Consider the iris dataset:

```
In [2]: iris = sns.load_dataset("iris")
```

```
In [3]: iris.head()
```

```
Out[3]:    sepal_length  sepal_width  petal_length  petal_width species
        0           5.1          3.5           1.4          0.2  setosa
        1           4.9          3.0           1.4          0.2  setosa
        2           4.7          3.2           1.3          0.2  setosa
        3           4.6          3.1           1.5          0.2  setosa
        4           5.0          3.6           1.4          0.2  setosa
```

There are only 5 features, namely:

```
In [4]: iris.columns
```

```
Out[4]: Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
               'species'],
              dtype='object')
```

Examining the species gives the following species type:

```
In [5]: iris["species"].unique()
```

```
Out[5]: array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

Let's now construct a new column that indicates if the species is setosa (using 1 to indicate positive identification and 0 otherwise):

```
In [6]: iris["is_setosa"] = (iris['species'] == "setosa")*1
```

```
In [7]: iris.head()
```

```
Out[7]:    sepal_length  sepal_width  petal_length  petal_width species  is_setosa
        0           5.1          3.5           1.4          0.2  setosa          1
        1           4.9          3.0           1.4          0.2  setosa          1
        2           4.7          3.2           1.3          0.2  setosa          1
        3           4.6          3.1           1.5          0.2  setosa          1
        4           5.0          3.6           1.4          0.2  setosa          1
```

And again the indicator for the virginica species:

```
In [8]: iris["is_virginica"] = (iris['species'] == "virginica")*1
```

We do the same for the versicolor species:

```
In [9]: iris["is_versicolor"] = (iris['species'] == "versicolor")*1
```

This is called **one-hot encoding**, where we create new indicator columns for all the values of the categorical variables.

0.1.2 Training and Testing

With every dataset we have limited data and wish to conserve as much of it as possible for training the model (using the data to give the model 'intelligence'). However, we need to consider how well our model performs on unobserved data points, which gives rise to the idea of splitting our data set in two, one to **train** our model and one to **test** our model:

First, take some observations randomly out of the dataset:

```
In [10]: idx = np.arange(iris.shape[0])
         np.random.seed(1)
         np.random.shuffle(idx)
```

We consider taking half the data set out for our test set:

```
In [11]: n = iris.shape[0]//2
         iris_test = iris.iloc[idx[:n]]
         iris_train = iris.iloc[idx[n:]]
         iris_test.head()
```

```
Out[11]:      sepal_length  sepal_width  petal_length  petal_width     species  \
         14            5.8          4.0           1.2          0.2      setosa
         98            5.1          2.5           3.0          1.1  versicolor
         75            6.6          3.0           4.4          1.4  versicolor
         16            5.4          3.9           1.3          0.4      setosa
         131           7.9          3.8           6.4          2.0   virginica

              is_setosa  is_virginica  is_versicolor
         14           1             0              0
```

```
        98             0              0               1
        75             0              0               1
        16             1              0               0
        131            0              1               0
```

```
In [12]: print(iris_train.shape)
         iris_train.head()
```

```
(75, 8)
```

```
Out[12]:      sepal_length  sepal_width  petal_length  petal_width     species  \
         74            6.4          2.9           4.3          1.3  versicolor
         116           6.5          3.0           5.5          1.8   virginica
         93            5.0          2.3           3.3          1.0  versicolor
         100           6.3          3.3           6.0          2.5   virginica
         89            5.5          2.5           4.0          1.3  versicolor

              is_setosa  is_virginica  is_versicolor
         74           0             0              1
         116          0             1              0
         93           0             0              1
         100          0             1              0
         89           0             0              1
```

We consider taking half the data set out for our test set:

```
In [13]: print(iris_test.shape)
         iris_test.head()
```

```
(75, 8)
```

```
Out[13]:      sepal_length  sepal_width  petal_length  petal_width     species  \
         14            5.8          4.0           1.2          0.2      setosa
         98            5.1          2.5           3.0          1.1  versicolor
         75            6.6          3.0           4.4          1.4  versicolor
         16            5.4          3.9           1.3          0.4      setosa
         131           7.9          3.8           6.4          2.0   virginica

              is_setosa  is_virginica  is_versicolor
         14           1             0              0
         98           0             0              1
         75           0             0              1
         16           1             0              0
         131          0             1              0
```

Let's try slicing the training data set by slicing it. Say we want to isolate the observations where `sepal_length` is greater than 4 and the `species` is Setosa :

(Note: We use binary operators `&` or `|` and bracketise the conditions because we need to do an element-wise comparison and `&` has a higher operator precedence than the comparator.)

```
In [14]: iris_slice = iris_train[(iris_train['sepal_length']>4) & (iris_train['species']=='seto
         iris_slice.head()
```

```
Out[14]:     sepal_length  sepal_width  petal_length  petal_width species  is_setosa  \
         10           5.4          3.7           1.5          0.2  setosa          1
         34           4.9          3.1           1.5          0.2  setosa          1
         32           5.2          4.1           1.5          0.1  setosa          1
         38           4.4          3.0           1.3          0.2  setosa          1
         27           5.2          3.5           1.5          0.2  setosa          1

             is_virginica  is_versicolor
         10             0              0
         34             0              0
         32             0              0
         38             0              0
         27             0              0
```

Now, let's train a regression model that predicts petal length using sepal length on the training data now:

```
In [15]: from sklearn.linear_model import LinearRegression

         model = LinearRegression()
```

Set the predictor to be the the `sepal_length` column of the iris training data set, our target is then set to the `petal_length` of the training set. We need to reshape it such that it becomes a $nx1$ column in order for it to be used in the Linear Regression model.

```
In [16]: predictor = iris_train['sepal_length'].values.reshape(-1,1)
         target = iris_train['petal_length']


         model.fit(predictor,target)

/usr/local/lib/python3.6/site-packages/sklearn/linear_model/base.py:509: RuntimeWarning: inter
  linalg.lstsq(X, y)


Out[16]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Observing our intercept and slope coefficients gives us:

```
In [17]: model.intercept_,model.coef_

Out[17]: (-7.114361148914037, array([1.87805789]))
```

Now, let's use this model to predict the petal length using the sepal length from the test data:

```
In [18]: sepal_test = iris_test['sepal_length'].values.reshape(-1,1)
         petal_test = iris_test['petal_length']

         predicted_petal = model.predict(sepal_test)
```

This gives us the predicted values of the petal_length, given by $\hat{y}_{predict}$, using our model. In order to evaluate our model, we want to compare the predicted values to the true values of petal length. Thus we need to consider the following error, $\epsilon_{predict}$:

$$\epsilon_{predict} = \hat{y}_{predict} - y_{obs}$$

Note that since the errors can be either positive or negative, one must square the errors in order to consider its magnitude.

To calculate the overall error, we can consider the sum of squared errors, denoted as SSE, which is given by:

$$SSE = \sum^{n} \epsilon^2_{predict}$$

```
In [19]: SSE = np.sum((predicted_petal-petal_test)**2)
         print(SSE)
```

```
61.72178946378996
```

The quantity above tells us the overall magnitude of errors, which may be meaningless as you would expect the SSE to grow with the number of samples. Scaling this gives us a more interpretable error. If we scale it by the number of data points, we can get an idea of the expected squared error of our prediction. Consider the mean squared error, denoted by MSE:

$$MSE = \frac{1}{n}\sum^{n} \epsilon^2_{predict}$$

```
In [20]: MSE = ((predicted_petal-petal_test)**2).mean()
         print(MSE)
```

```
0.8229571928505328
```

The Root Mean Squared Error, $RMSE$, which will give you an error of the same scale as your data is given by:

$$RMSE = \sqrt{MSE}$$

```
In [21]: RMSE = np.sqrt(MSE)
         print(RMSE)
```

```
0.9071698809211717
```

0.1.3 Model Selection

In our case above, we have a number of parameters that we can consider when building our model, but what is the set of parameters that gives us the best (lowest) $RMSE$?

Recall that we have the following features:

```
In [22]: iris.columns
```

```
Out[22]: Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species',
               'is_setosa', 'is_virginica', 'is_versicolor'],
              dtype='object')
```

The number of all possible subsets of the features number at $2^{\# of\ features}$, which is 64 possibilities for this instance. This would not be feasible if the number of features is large, thus, we might defer to the following method of selecting the best set:

Greedy Forward Selection Doing the regression step and calculating the RMSE for each of the features gives:

We have the following function to calculate RMSE for each particular feature:

```
In [34]: def RMSE_Petal_Length(feature):
             model = LinearRegression()
             predictor = iris_train[feature].values.reshape(-1,1)
             target = iris_train['petal_length']
             model.fit(predictor,target)
             predictor_test = iris_test[feature].values.reshape(-1,1)
             target_test = iris_test['petal_length']
             predicted_value = model.predict(predictor_test)

             RMSE = np.sqrt(((target_test - predicted_value)**2).mean())
             return RMSE
```

Now, let's construct the vector of features with which we can use to calculate RMSE:

```
In [35]: feature_set = pd.Series(iris.drop(['species','petal_length'],axis=1).columns)
         feature_set

Out[35]: 0     sepal_length
         1      sepal_width
         2      petal_width
         3        is_setosa
         4     is_virginica
         5    is_versicolor
         dtype: object
```

Applying the RMSE function gives:

```
In [44]: feature_set_RMSE = feature_set.apply(RMSE_Petal_Length)
         feature_set_RMSE = pd.concat([feature_set,feature_set_RMSE],axis = 1)
         feature_set_RMSE.columns = ["Feature_Name","RMSE"]
         feature_set_RMSE

Out[44]:     Feature_Name      RMSE
         0   sepal_length  0.907170
         1    sepal_width  1.595930
         2    petal_width  0.485488
         3      is_setosa  0.705394
         4   is_virginica  1.231345
         5  is_versicolor  1.783849
```

From the set of RMSEs above, we can see that the best among the 1-feature predictors is the `petal_width` predictor. Now, given this set of information let's do the regression again with some combination of the `petal_width` and another predictor.

```
In [46]: def RMSE_Petal_Length_w_petal_width(feature):
             model = LinearRegression()
             predictor = iris_train[[feature,'petal_width']]
             target = iris_train['petal_length']
             model.fit(predictor,target)
             predictor_test = iris_test[[feature,'petal_width']]
             target_test = iris_test['petal_length']
             predicted_value = model.predict(predictor_test)

             RMSE = np.sqrt(((target_test - predicted_value)**2).mean())
             return RMSE
```

```
In [48]: feature_set2 = pd.Series(iris.drop(['species','petal_length','petal_width'],axis=1).co
         feature_set2_RMSE = feature_set2.apply(RMSE_Petal_Length_w_petal_width)
         feature_set2_RMSE = pd.concat([feature_set2,feature_set2_RMSE],axis = 1)
         feature_set2_RMSE.columns = ["Feature_Name","RMSE"]
         feature_set2_RMSE
```

```
Out[48]:     Feature_Name      RMSE
         0   sepal_length  0.421607
         1    sepal_width  0.473316
         2      is_setosa  0.426598
         3   is_virginica  0.482184
         4  is_versicolor  0.465793
```

We see that a combination of the `petal_width` and the `sepal_length` features produces a prediction RMSE of 0.42, which is a decrease of 0.06.

The idea of **greedy forward selection** follows from this as one seeks to minimise the model's RMSE by progressively selecting features (in a greedy fashion) until the RMSE can no longer be reduced.

Greedy Backward Selection **Greedy backward selection** is a paradigm of model selection much like forward selection, only that you start from all the features and progressively exclude the number of features until RMSE cannot be reduced further.

We first calculate the RMSE for the model with ALL features present:

```
In [49]: model = LinearRegression()
         predictor = iris_train.drop(['species','petal_length'],axis=1)
         target = iris_train['petal_length']
         model.fit(predictor,target)
         predictor_test = iris_test.drop(['species','petal_length'],axis=1)
         target_test = iris_test['petal_length']
         predicted_value = model.predict(predictor_test)

         RMSE = np.sqrt(((target_test - predicted_value)**2).mean())
         RMSE
```

```
Out[49]: 0.3005272052390222
```

```
In [71]: def bs_RMSE_Petal_Length(feature):
             model = LinearRegression()
             predictor = iris_train.drop(['species','petal_length',feature],axis=1)
             target = iris_train['petal_length']
             model.fit(predictor,target)
             predictor_test = iris_test.drop(['species','petal_length',feature],axis=1)
             target_test = iris_test['petal_length']
             predicted_value = model.predict(predictor_test)

             RMSE = np.sqrt(((target_test - predicted_value)**2).mean())
             return RMSE
```

```
In [75]: back_feature_set = pd.Series(iris.drop(['species','petal_length'],axis=1).columns)
         back_feature_set_RMSE = back_feature_set.apply(bs_RMSE_Petal_Length)
         back_feature_set_RMSE = pd.concat([back_feature_set,back_feature_set_RMSE],axis = 1)
         back_feature_set_RMSE.columns = ["Feature_Name","RMSE"]
         back_feature_set_RMSE
```

```
Out[75]:      Feature_Name      RMSE
         0    sepal_length  0.396753
         1     sepal_width  0.300739
         2     petal_width  0.318032
         3       is_setosa  0.300527
         4    is_virginica  0.300527
         5   is_versicolor  0.300527
```

We can see that eliminating one of the categorical variables `is_setosa`, `is_virginica`, or `is_versicolor`, allows the RMSE of the model to remain the same. In general we would favour a model with fewer parameters as that would mean a more simplistic model that is more likely to generalise well. Repeating this process until one can no longer keep the RMSE as small as possible leads to the model being selected.

0.1.4 Logistic Regression

Up until now, we learnt about the process of regressing a continous variable as the target against a set of continuous and categorical variables. But what if we find ourselves in the situation where the desired target variable is categorical in nature?

We first consider the case where the target variable is binary valued:

$$y \in \{0,1\}$$

By considering a set of continuous and categorical variables, we want to predict y.

In the case of the iris dataset, let y be the `is_setosa` indicator variable. Consequently, fitting the logistic regression model is similar to before:

```
In [76]: from sklearn.linear_model import LogisticRegression
         model = LogisticRegression()
         predictor = iris_train.drop(['species','is_setosa','is_virginica'],axis=1)
         target = iris_train['is_setosa']

         model.fit(predictor,target)

Out[76]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

We then use the model to try to predict whether a plant is of the setosa species based on the other variables. Bear in mind that the predictions are now binary valued.

```
In [79]: predictor_test = iris_test.drop(['species','is_setosa','is_virginica'],axis=1)
         target_test = iris_test['is_setosa']
         predicted_value = model.predict(predictor_test)
         predicted_value

Out[79]: array([1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0,
                0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0,
                0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0,
                1, 0, 0, 0, 0, 0, 0, 1, 0])
```

We then compare our model prediction to the actual observed value of the target. Because the variable is binary in nature, we see that the errors are also binary valued. This then tells us that the SSE is equivalent to the sum of the absolute errors, therefore we can interpret the MSE as the proportion correctly prediction instances out of the total number of observations. This is known as the **classification error**.

```
In [80]: sum(np.abs(target_test - predicted_value))/len(target_test)

Out[80]: 0.0
```