Worksheet 2 Week 14

Martha Lewis

Introduction

This worksheet covers the three supervised learning algorithms we looked at in week 14: k-nearest neighbours, linear regression, and the naive Bayes classifier. Similar to last week, you will do some work implementing your own versions of these algorithms, to ensure that you understand the details of them. You will also compare them with the implementations in scikit-learn to test your implementations.

Preliminaries

Import key packages: NumPy, matplotlib, and any others that you prefer to work with. In general, when writing code, you will put all your import statements at the top. However, for these worksheets we will import as we go along.

[ ]:
```
#TODO: import NumPy and matplotlib here
```

Question 1: k-nearest neighbours classification

In this question we will use the k-nearest neighbours algorithm to make predictions on the breast cancer Wisconsin dataset. This is a classification problem where the aim is to classify instances as either being malignant or benign based on the following 10 features:

1. radius (mean of distances from center to points on the perimeter)
2. texture (standard deviation of gray-scale values)
3. perimeter
4. area
5. smoothness (local variation in radius lengths) 6. compactness (perimeter squared/ area -1)
6. concavity (severity of concave portions of the contour)
7. concave points (number of concave portions of the contour)
8. symmetry
9. fractal dimension ('coastline approximation' -1)

In this question you will (a) download the dataset from sklearn and store the data and targets in suitable variables, (b) separate your data into a training and test split, (c) write your own function to implement k-nearest neighbours, (d) check your implementation with that of sklearn. We then go on to (e) select the most appropriate value of k using cross-validation.

Part (a)

Import the package `datasets` from `sklearn` and then load the breast cancer dataset (function is `load_breast_cancer()`). Save the data into a variable `X` and the targets into a variable `Y`. Take a look at the data in `X`. How many datapoints are there? How many features does each datapoint have? (Hint: use `np.shape`). Take a look at the targets. Is this suitable for a classification algorithm or a regression algorithm?

[ ]:
```
# Import suitable packages, load the dataset, and save data and targets
# into variables X and Y.
# import packages
##TODO##

# load dataset and save data and targets into X and Y
##TODO##
```

Part (b)

Use the function `train_test_split` from `sklearn.model_selection` to split your data into a training set and a held-out test set. Use a test set that is 0.2 of the original dataset. Set the parameter `random_state` to 10 to help with replication.

[ ]:
```
# Import the package train_test_split from sklearn.model_selection.
# Split the dataset into Xtr, Xtest, Ytr, Ytest. Xtest and Ytest will form
# your held-out test set.
# You will later split Xtr and Ytr into training and validation sets.
Xtr, Xtest, Ytr, Ytest = ##TODO##
```

Part (c)

Recall from the lecture that the k-nearest neighbours algorithm runs as follows:

Training step: Simply store the dataset

Prediction step: Given a datapoint $\vec{x}$:

- **Find** the k datapoints $(\vec{x}_i, y_i)$ where the distance from $\vec{x}$ to $\vec{x}_i$ is smallest
- **Return** the majority class from the y_i

What, if anything, do you need to do for the training step?

Write function(s) to implement the k-nearest neighbours prediction step. You may wish to break the procedure down into two functions `predict_datapoint` that makes a prediction for one datapoint and `predict_data` that loops over the whole dataset.

To select the majority class from the nearest neighbours, you can use the function `scipy.stats.mode()`

[ ]:
```
from scipy.stats import mode
# Write function(s) to implement the prediction step in k-nearest neighbours.
# You can use the suggested structure below if desired.


# predict_datapoint takes 4 arguments.
# pt (type: numpy array) is the datapoint we are making a prediction about,
# Xtrain and Ytrain (numpy arrays) are training data and targets,
# k (int) is the number of neighbours.
# Returns an integer which is the predicted class for pt
def predict_datapoint(pt, Xtrain, Ytrain, k):
    # For each datapoint in Xtrain, calculate the distance to pt and store
    ##TODO##

    # Sort the list of distances (hint: use np.argsort)
    ##TODO##

    # obtain the k classes (in Ytrain) of the datapoints with the
    # smallest distance to pt
    ##TODO##

    # return the mode of the classes
    ##TODO##

# predict_data takes 4 arguments:
# the test data Xtst (numpy array),
# the training data Xtrain (numpy array),
# the training targets Ytrain (numpy array),
# the number of neighbours k (int, default = 3).
# Returns: predictions (array of int) for each point in Xtst
def predict_data(Xtst, Xtrain, Ytrain, k=3):
    #Loop over the datapoints in Xtst and store the prediction for that datapoint
    ##TODO##
    # Return the predictions
    ##TODO##

# Predict values for the TRAINING data (we will not look at the test set yet)
##TODO##
```

Part (d)

Now we can compare your implementation with the sklearn implementation (you should get the same results). Firstly import the classfifier `KNeighborsClassifier` from `sklearn.neighbors`. Instantiate the classifier with the same number of neighbours that you used previously. Fit the model and make a prediction on the test set.

[ ]:
```
# Import KNeighborClassifier
##TODO##

# Instantiate the classifier with 3 neighbors
##TODO##

#Fit the classifier on the training data
##TODO##

#Make a prediction on the training data
##TODO##
```

Check whether your predictions are the same as the predictions from `KNeighborsClassifier`.

[ ]:
```
##TODO##
```

Use the built in metrics in sklearn to calculate the accuracy of your classifier on the TRAINING set.

[ ]:
```
##TODO##
```

Part(e) Using cross-validation for model selection

k-nearest neighbours has the parameter k, and we need to decide which is the best value of k to use. Last week we talked about using cross-validation for model selection.

We will use cross-validation on our training set to select the best value of k, in a range from 1 to 30.

NB: use sklearn's version of k-NN rather than yours, since unless you have optimised yours it is probably too slow.

Since we are using cross-validation for model selection we will cross-validate on the training set only.

Procedure:

1. Import `KFold` from `sklearn.model_selection`
2. Instantiate `KFold` with 5 splits. Set the parameter `random_state` to help you reproduce your results if needed.
3. Set a variable `max_k` to 30

4. Inititalise two variables to store the training accuracies and validation accuracies (these need to store max_k*5 accuracies)

5. Loop over the values of k:
    1. Instantiate a k-nn classifier (Use the sklearn classifier) with the current value of k

    2. Loop over the cross-validation splits:
        1. fit the model on the current split of data

        2. make predictions

        3. calculate training and validation accuracy and store

6. Calculate the mean training and validation accuracies across splits for each k

Plot the mean training and validation accuracies. Which value of k will you use? Why?

[ ]:
```
# Use cross-validation to select the value of k
# You can use the structure below if desired

# Import KFold from sklearn.model_selection
##TODO##

# Instantiate KFold with 5 splits.
# Set the parameter random_state to help you reproduce your results if needed.
##TODO##

# Set a variable max_k to 30
##TODO##

# Inititalise two variables to store the
# training accuracies and validation accuracies
# (these need to store max_k*5 accuracies)
##TODO##

# Loop over the values of k:
for k in range(max_k):

    # Instantiate a k-nn classifier (Use the sklearn classifier)
    # with the current value of k
    ##TODO##

    # Loop over the cross-validation splits:
    ##TODO##

        # fit the model on the current split of data
        ##TODO##
```

```
        # make predictions
        ##TODO##

        # calculate training and validation accuracy and store
        ##TODO##

# Calculate the mean training and validation accuracies across splits for each k
##TODO##
```

[ ]:
```
# Plot the mean training and validation accuracies against each value of k.
#  Which value of k will you use? Why?
##TODO##
```

Question 2: The naive Bayes classifier

Recall from the lecture notes that the naive Bayes classifier works as follows. We are trying to approximate an unknown function

$$f : \Omega \to \mathcal{O}$$

where Ω is our feature space and our output space $\mathcal{O} = \{c_1, c_2, ...c_K\}$ is a finite set of classes.

The naive Bayes classifier does this by building a model that assigns the class label $\hat{y} = c_k$ as follows:

$$\hat{y} = \text{argmax}_k p(c_k) \prod_i p(x_i|c_k)$$

i.e., the k that maximises this quantity.

In practice, multiplying all the $p(x_i|c_k)$ together is going to give some very small values. Therefore, we can take the log to make it easier to compute:

$$\hat{y} = \text{argmax}_k p(c_k) \prod_i p(x_i|c_k) = \text{argmax}_k log(p(c_k) \prod_i p(x_i|c_k)) \tag{1}$$

$$= \text{argmax}_k log(p(c_k)) + \sum_i log(p(x_i|c_k)) \tag{2}$$

If we choose that $p(x_i|c_k)$ is given by a normal distribution with mean μ_k and variance σ_k^2, then we obtain the following expression:

$$\hat{y} = \text{argmax}_k \log(p(c_k)) + \sum_i \log(p(x_i|c_k)) \tag{3}$$

$$= \text{argmax}_k \log(p(c_k)) + \sum_i \log\left(\frac{1}{\sigma_k\sqrt{2\pi}} exp\left(\frac{-(x-\mu_k)^2}{2\sigma_k}\right)\right) \tag{4}$$

$$= \text{argmax}_k \log(p(c_k)) - \sum_i \log\left(\sigma_k\sqrt{2\pi}\right) - \sum_i \left(\frac{(x-\mu_k)^2}{2\sigma_k}\right) \quad \text{log-likelihood} \tag{5}$$

Expressing the values in terms of these sums means that they do not get so small, and it is less likely that there will be errors at the machine precision level.

How do we implement this in practice? We assume that each probability $p(x_i|c_k)$ is given by some distribution, and then given a datapoint $\vec{x}$, we plug the value into the equation for the distribution.

In this question you will (a) implement your own version of the Gaussian naive Bayes classifier, (b) check your classifier against the implementation in sci-kit learn, (c) compare the accuracy of the naive Bayes classifier with the accuracy of the k-nearest neighbours classifier, and (d) run cross-validation to verify whether the kNN classfier or the Gaussian naive Bayes classifier performs better on this dataset.

Part (a) Implementing Gaussian naive Bayes

For this question we will make the assumption that each feature is described by a normal (also called Gaussian) distribution. The procedure is as follows: 1. Divide the training data by class 2. Calculate mean and standard deviation per class and per feature 4. For each datapoint in the validation set, calculate the log-likelihood for each class and for each feature (Hint: use the function `scipy.stats.norm.logpdf`) 5. Combine these values together with the probability of the class according to the log-likelihood equation above 6. Choose the class with the highest value

[ ]:
```
##TODO##
# Write your own implementation of naive Bayes applied to the breast cancer␣
 ↪dataset.

# If you wish you can follow the structure below
from scipy.stats import norm

# Split training data Xtr into training and validation with an 80:20 split.
# Set the random state to help with reproducibility
##TODO##

#Separate the training set into classes, one set of data for each class
##TODO##

# Calculate the means and standard deviations for each class, for each feature.
# There are 30 features in the dataset, so you should have a 30-dimensional
# array of means for each class and a 30-dimensional array of standard deviations
# for each class. Remember that you can take the mean across rows or columns of
# a matrix by specifying axis = 1 or axis = 0
##TODO##

# Calculate the prior probability p(c_i) for each class
##TODO##

# Calculate the log-likelihood of each class for each datapoint in the␣
 ↪validation set
```

```
# Hint: you can use the function scipy.stats.norm.logpdf to help with this
##TODO##

# Your predicted class is 0 if class 0 has the highest log-likelihood,
# and 1 if class 1  has the highest log-likelihood
##TODO##
```

Part (b) Checking results

We now compare our results with the sklearn implementation.

[ ]:
```
##Import the classifier GaussianNB from sklearn.naive_bayes
from sklearn.naive_bayes import GaussianNB
# Instantiate the classifier (use the parameter var_smoothing=0.0),
# fit, and predict the classes
##TODO##
```

[ ]:
```
# Compare your predicted classes with those of the sklearn implementation.
# If they are not identical, this may be due to some differences in parameters.
# They should be almost all the same, however.
##TODO##
```

Part (c) Comparing k-nearest neighbours and Gaussian naive Bayes

Now retrain the naive Bayes classifier using the original training set `Xtr`, `Ytr`. Also, retrain the k-nearest neighbours classifier using `Xtr` and `Ytr`. Use the value of k that you decided on using cross-validation. You can use the sklearn implementations of knn and naive Bayes.

Compute the accuracy of the naive Bayes classifier over the training set and the held-out test set.

Compare with the accuracy of the k-nearest neighbours classifier on each set.

Is it clear which classifier is the best on this dataset? why or why not?

[ ]:
```
# Instantiate the knn classifer with your chosen value of k
##TODO##

# Fit the Gaussian naive Bayes classifier and the knn classifier on Xtr, Ytr
##TODO##

# Make predictions for the training set and the test set
##TODO##

# Take a look at the accuracy scores
##TODO##
```

Part (d) Using cross-validation for statistical validation

Earlier we used cross-validation to select the model parameters we would be using. We can also use it another way: to provide statistical information about which model is best. We will set up cross-validation on the whole dataset, with 10 folds.

- Compute the accuracy for each model on the test set on each fold.
- Calculate the mean accuracy across folds. Which model performs best?
- Make a box-plot of the spread of scores of each model. Is there a clear difference between model performance?
- Perform a paired t-test on the accuracy scores. What can you conclude about the performance of the two models?

[ ]:
```
# Set up a k-fold cross-validation with 10 folds
##TODO##
```

[ ]:
```
# For each fold, fit each model on the training data
# and compute accuracy on the test data.
##TODO##
```

[ ]:
```
# Compute the mean and standard deviation of the accuracies for each model.
# Does one model perform better?
##TODO##
```

[ ]:
```
# Make a boxplot of the accuracy scores. (Use plt.boxplot).
# Is there a clear difference between the models?
##TODO##
```

[ ]:
```
# Perform a paired t-test (you can use the function scipy.stats.ttest_rel).
# What do you conclude about the performance of the two models?
##TODO##
```

Question 3: Linear Regression

In linear regression we make the assumption that the data (x_i, y_i) can be modelled by a function of the form

$$\hat{y}_i = f(\vec{x}_i) = \sum_j a_j x_{ij} + b_i$$

Recall that we can express this in a matrix format by:

$$\hat{\vec{y}} = f(X) = X\Theta$$

where

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} & 1 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,n} & 1 \end{pmatrix}, \quad \vec{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}, \quad \Theta = \begin{pmatrix} a_1 \\ \vdots \\ a_n \\ b \end{pmatrix}$$

We saw in lectures that the optimal value of Θ is given by setting

$$\Theta = (X^T X)^{-1} X^T \vec{y}$$

The quantity $(X^T X)^{-1} X^T$ is called the psuedoinverse of X, and can be computed using the function `np.linalg.pinv`.

We will (a) perform a linear regression on the diabetes dataset. You can load this dataset using the function `load_diabetes` from `sklearn.datasets`. (b) compute the mean squared error and the R^2, and (c) compare your results with the built in function in sklearn (`sklearn.linear_model.LinearRegresion()`). You should get the same results.

[ ]:
```
# import statments here
from sklearn import datasets, linear_model
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

Part (a) Implementing linear regression

[ ]:
```
# Load the diabetes dataset
##TODO##

# Split the dataset into training and test, using test_size=0.2
##TODO##
```

[ ]:
```
# Add a column of ones to Xtrain and Xtest for the intercept term
##TODO##
```

[ ]:
```
# Calculate the value of the coefficients theta.
# You can use the function np.linalg.pinv
##TODO##
```

Part (b) Computing performance metrics

[ ]:
```
# Make a prediction on the test set using the coefficients theta
##TODO##
```

[ ]:
```
# Calculate the mean squared error and the R^2.
# You can use the built in functions from sklearn
##TODO##
```

Part (c) Checking results

Compare your results with the built in function `sklearn.linear_model.LinearRegression()`

[ ]:
```
# Instantiate the linear regression
##TODO##
```

[ ]:
```
# Fit the model and make a prediction on the test set.
# Compare with your implementation
##TODO##
```

Visualise the perfomance of the regression by plotting your predicted values vs target values on a scatter plot, and drawing a line y=x. If all predictions were perfect, the predicted values would lie on the line.

[ ]:
```
# Plot predicted values vs target values on a scatter plot, and draw line y=x
## TODO##
```

Extra question: Polynomial regression

The term 'linear' in linear regression refers only to the coefficients Θ. We can in fact compute polynomial terms in the data and perform linear regression over this extended dataset to get a better fit to the data.

To compute polynomial terms in the data automatically, you can use the class `sklearn.preprocessing.PolynomialFeatures`. To find out how to use it, look at the guidance (you can type `help(PolynomialFeatures)` once you have imported it).

The following small dataset (in the cell below) gives a relationship between temperature and yield for an experiment. Use cross-validation to select the degree of the polynomial that best fits this data.

Plot the mean squared error against degree on the training set and on the validation set. Which degree of polynomial best fits this data?

[ ]:
```
# Data
X = np.array([50,50,50,70,70,70,80,80,80,90,90,90,100,100,100]).reshape(-1, 1)
y = np.array([3.3,2.8,2.9,2.3,2.6,2.1,2.5,2.9,2.4,3,3.1,2.8,3.3,3.5,3]).
 ↪reshape(-1, 1)
```

[ ]:

[ ]:

[ ]: