Chapter 1 - Classification

k-Nearest Neighbors Classification

Train a KNN model and predict labels of new data points

In [1]:
```
# Import dependencies
from sklearn.neighbors import KNeighborsClassifier
import pandas as pd
import numpy as np
```

In [2]:
```
# Load data
churn_df = pd.read_csv('./datasets/telecom_churn_clean.csv', index_col=[0])
```

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

Out[3]:

	account_length	area_code	international_plan	voice_mail_plan	number_vmail_messages	total_day_mir
0	128	415	0	1	25	
1	107	415	0	1	26	
2	137	415	0	0	0	
3	84	408	1	0	0	
4	75	415	1	0	0	

In [4]:
```
# Create arrays for the features and the target variable
y = churn_df["churn"].values
X = churn_df[["account_length", "customer_service_calls"]].values
```

In [5]:
```
# Create a KNN classifier with 6 neighbors
knn = KNeighborsClassifier(n_neighbors=6)
```

In [6]:
```
# Fit the classifier to the data
knn.fit(X, y)
```

Out[6]:
```
KNeighborsClassifier(n_neighbors=6)
```

In [7]:
```
X_new = np.array([[30.0, 17.5],
                  [107.0, 24.1],
                  [213.0, 10.9]])
```

In [8]:
```
# Predict the labels for the X_new
y_pred = knn.predict(X_new)
```

In [9]:
```
# Print the predictions for X_new
print("Predictions: {}".format(y_pred))
```

```
Predictions: [0 1 0]
```

The first and third customers were predicted to not churn.

Train/test split and computing accuracy

In [10]:
```
# Import dependencies
from sklearn.model_selection import train_test_split
```

In [11]:
```
X = churn_df.drop("churn", axis=1).values
y = churn_df["churn"].values
```

In [12]:
```
# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)
knn = KNeighborsClassifier(n_neighbors=5)
```

In [13]:
```
# Fit the classifier to the training data
knn.fit(X_train, y_train)
```

Out[13]:
```
▾ KNeighborsClassifier

KNeighborsClassifier()
```

In [14]:
```
# Print the accuracy
print(knn.score(X_test, y_test))
```

```
0.8740629685157422
```

We get an accuracy of 87%.

Overfitting and underfitting

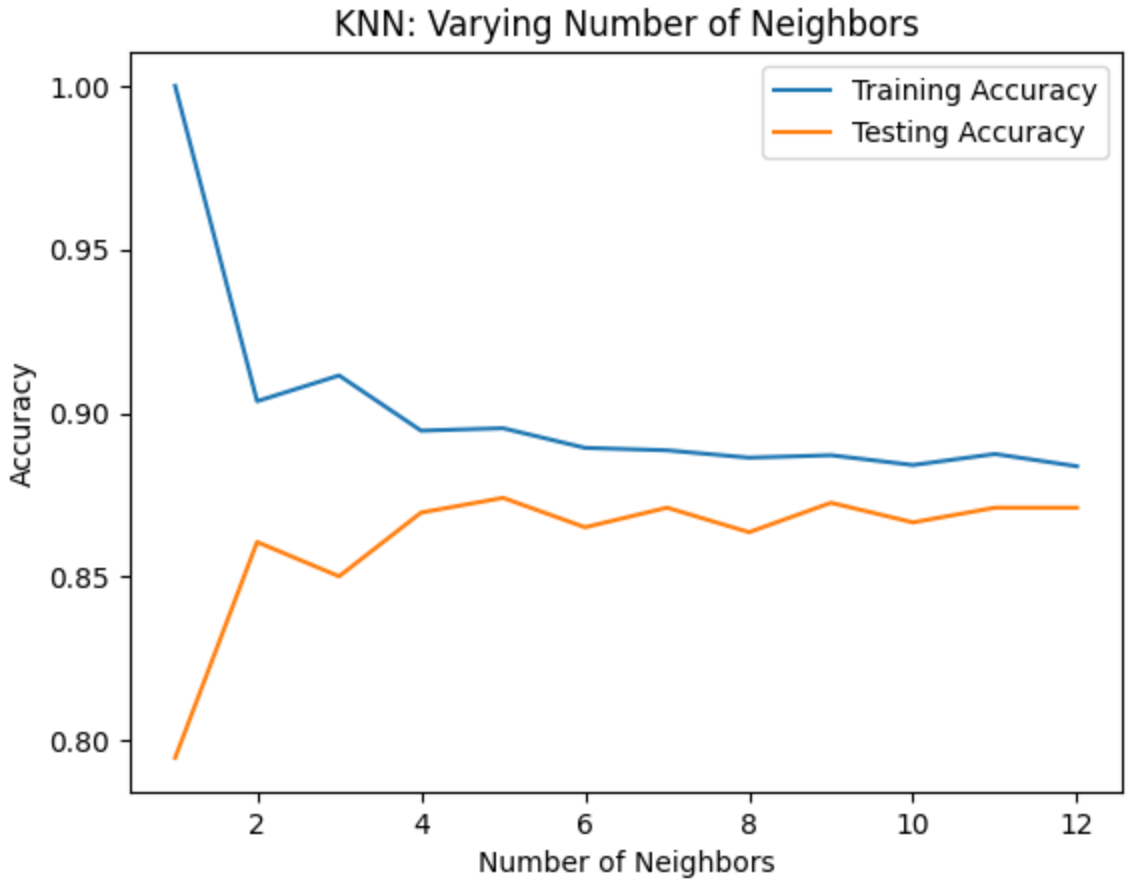
In [15]:
```
# Import dependencies
import matplotlib.pyplot as plt
```

In [16]:
```
# Create neighbors
neighbors = np.arange(1, 13)
train_accuracies = {}
test_accuracies = {}
```

In [17]:
```
for neighbor in neighbors:
    # Set up a KNN Classifier
    knn = KNeighborsClassifier(n_neighbors=neighbor)

    # Fit the model
    knn.fit(X_train, y_train)

    # Compute accuracy
    train_accuracies[neighbor] = knn.score(X_train, y_train)
    test_accuracies[neighbor] = knn.score(X_test, y_test)
```

In [18]:
```
# Visualise model complexity
plt.title("KNN: Varying Number of Neighbors")

# Plot training accuracies
plt.plot(neighbors, train_accuracies.values(), label="Training Accuracy")
# Plot test accuracies
```

```
plt.plot(neighbors, test_accuracies.values(), label="Testing Accuracy")

plt.legend()
plt.xlabel("Number of Neighbors")
plt.ylabel("Accuracy")
plt.show()
```



Training accuracy decreases as the number of neighbors gets larger (and vice versa for test accuracy).

For the test set, accuracy peaks with 7 neighbors suggesting it is the optimal value for our model.

In [ ]:

Chapter 2 - Regression

Creating features

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
sales_df = pd.read_csv('./datasets/advertising_and_sales_clean.csv')
```

sales_df

In [3]:
```
# Create X from the radio column's values
X = sales_df["radio"].values.reshape(-1, 1)

# Create y from the sales column's values
y = sales_df["sales"].values
```

In [4]:
```
X.shape, y.shape
```

Out[4]:
```
((4546, 1), (4546,))
```

Linear regression model

Simple linear regression: $y = ax + b$ (y: target, x: single feature, a, b: parameters of the model - slope, intercept)

Need to choose a line that minimises the error function/loss function/cost function (which is defined for any given line)

Residual: difference between line and an individual data point

Ordinary Least Squares (OLS): minimise $RSS = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

In [5]:
```
from sklearn.linear_model import LinearRegression
```

In [6]:
```
reg = LinearRegression()
reg.fit(X, y)
```

Out[6]:
```
▾ LinearRegression

LinearRegression()
```

In [7]:
```
predictions = reg.predict(X)
predictions[:5]
```

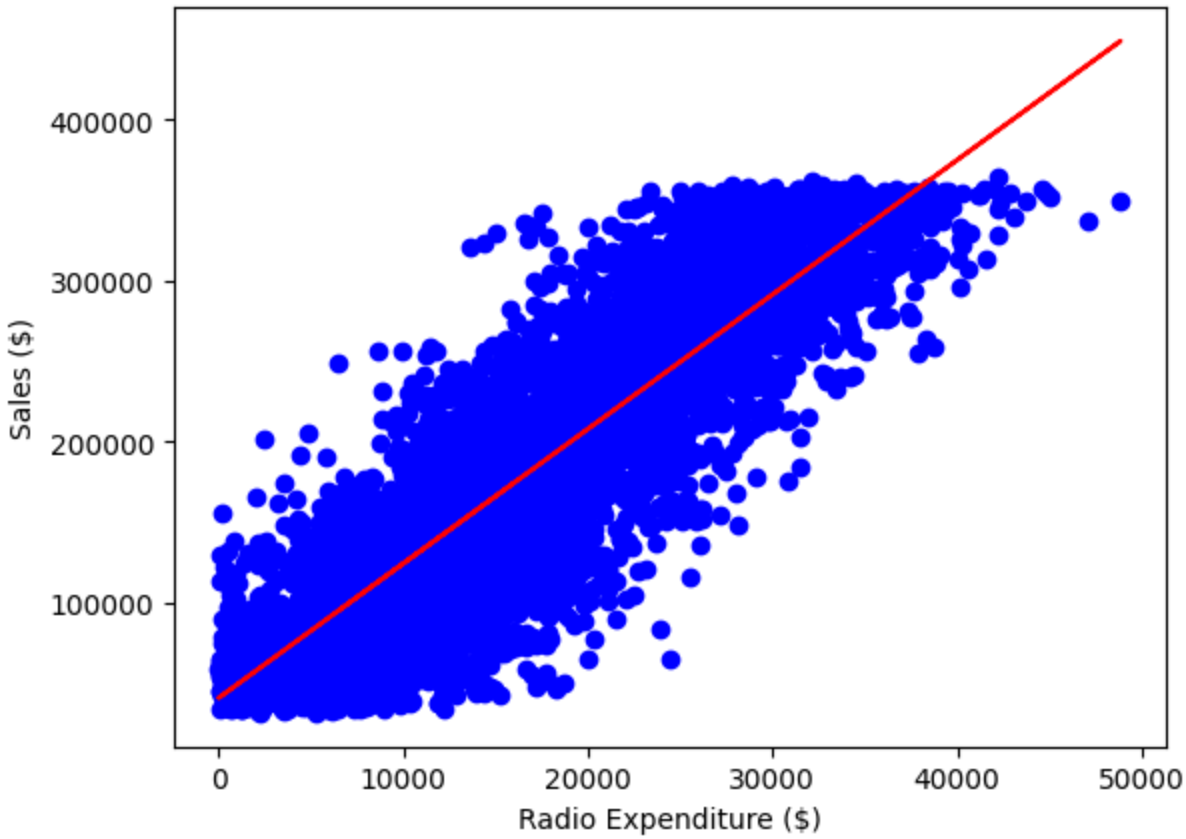
Out[7]:
```
array([ 95491.17119147, 117829.51038393, 173423.38071499, 291603.11444202,
       111137.28167129])
```

First 5 predictions range from 95,000 to over 290,000.

Visualise linear regression model

In [8]:
```
import matplotlib.pyplot as plt
```

In [9]:
```
# Create scatter plot
plt.scatter(X, y, color="blue")

# Create line plot
plt.plot(X, predictions, color="red")
plt.xlabel("Radio Expenditure ($)")
plt.ylabel("Sales ($)")
plt.show()
```



Near perfect correlation between radio advertising expenditure and sales.

Multiple linear regression

e.g. $y = a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n + b$

In [10]:
```
from sklearn.model_selection import train_test_split
```

In [11]:
```
# Create X and y arrays
X = sales_df.drop(["sales", "influencer"], axis=1).values
y = sales_df["sales"].values
```

In [12]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42
```

In [13]:
```
reg = LinearRegression()
reg.fit(X_train, y_train)
```

Out[13]:
```
▾ LinearRegression
LinearRegression()
```

In [14]:
```
# Make predictions
y_pred = reg.predict(X_test)
print("Predictions: {}, Actual Values: {}".format(y_pred[:2], y_test[:2]))
```

```
Predictions: [53176.66154234 70996.19873235], Actual Values: [55261.28 67574.9 ]
```

First 2 predictions appear to be within around 5% of the actual values from the test set.

Regression performance

R^2: quantifies the variance in target values explained by the features (0-1)

$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ (measured in target units, squared)

$RMSE = \sqrt{MSE}$ (measure $RMSE$ in same units as target variable)

In [15]:
```
from sklearn.metrics import mean_squared_error
```

In [16]:
```
r_squared = reg.score(X_test, y_test) # computes R-squared
```

In [17]:
```
rmse = mean_squared_error(y_test, y_pred, squared=False) # rmse (note squared=False)
```

In [18]:
```
# Print the metrics
print("R^2: {}".format(r_squared))
print("RMSE: {}".format(rmse))
```

```
R^2: 0.9990152104759368
RMSE: 2944.4331996001
```

Features explain 99.9% of the variance in sales values, hence the company's advertising strategy seems to be working well.

Cross-validation for R-squared

Use cross-validation: model performance dependent on way we split the data (not representative of model's ability to generalise to unseen data)

More folds = more computationally expensive

In [19]:
```
from sklearn.model_selection import cross_val_score, KFold
```

In [20]:
```
kf = KFold(n_splits=6, shuffle=True, random_state=5)
```

In [21]:
```
reg = LinearRegression()
```

In [22]:
```
cv_scores = cross_val_score(reg, X[:, [1, 2]], y, cv=kf) # computes 6-fold cross-validat
```

In [23]:
```
cv_scores
```

Out[23]:
```
array([0.74451678, 0.77241887, 0.76842114, 0.7410406 , 0.75170022,
       0.74406484])
```

For the chosen features, R-squared for each fold ranged between 0.74 and 0.77. Cross-validation helped us see how performance varies depending on how the data is split.

In [24]:
```
print("Mean:", np.mean(cv_scores))
print("Standard deviation:", np.std(cv_scores))
print("95% confidence interval: ", np.quantile(cv_scores, [0.025, 0.975]))
```

```
Mean: 0.7536937414361207
Standard deviation: 0.012305389070474737
95% confidence interval:  [0.74141863 0.77191916]
```

We get an average score of 0.75 with a low standard deviation.

Regularised regression: Ridge

Linear regression minimises a loss function (chooses coefficient a for each feature variable plus b)

Large coefficients can lead to overfitting

Regularisation penalises large +ve or -ve coefficients

Ridge regression performs regularisation by computing the squared values of the model parameters multiplied by alpha and adding them to the loss function

Ridge regression loss function = OLS loss function + $\alpha * \sum_{i=1}^{n} a_i^2$ (need to choose α hyperparameter)

$\alpha = 0$ = OLS (can lead to overfitting)

Very high α: can lead to underfitting

In [25]:
```
from sklearn.linear_model import Ridge
```

In [26]:
```
alphas = [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]
ridge_scores = []
```

In [27]:
```
for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train, y_train)
    score = ridge.score(X_test, y_test) # obtain R-squared
    ridge_scores.append(score)
```

In [28]:
```
ridge_scores
```

Out[28]:
```
[0.9990152104759369,
 0.9990152104759373,
 0.9990152104759419,
 0.999015210475987,
 0.9990152104764387,
 0.9990152104809561]
```

The scores don't seem to change much as alpha increases, which is indicative of how well the features explain the variance in the target. Even by penalising large coefficients, underfitting does not occur.

Lasso regression for feature importance

Lasso regression loss function = OLS loss function + $\alpha * \sum_{i=1}^{n} |a_i|$

Can use same sklearn code as ridge regression for lasso regression

Lasso can select important features of a dataset

Lasso shrinks coefficients of less important features to 0

Features not shrunk to 0 are selected by lasso

In [29]:
```
from sklearn.linear_model import Lasso
```

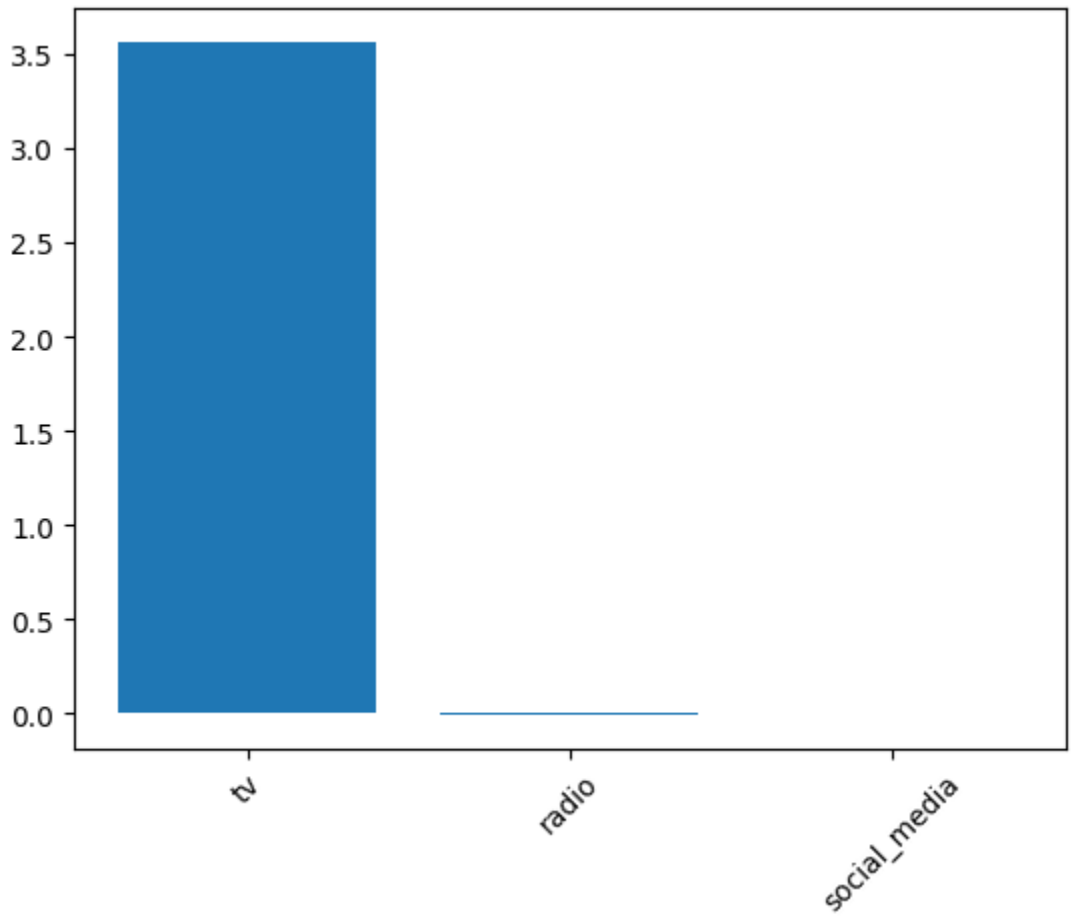
In [30]:
```
lasso = Lasso(alpha=0.3)
lasso.fit(X, y)
lasso_coef = lasso.coef_
```

In [31]:
```
lasso_coef
```

Out[31]:
```
array([ 3.56256962, -0.00397035,  0.00496385])
```

In [32]:
```
sales_columns = ['tv', 'radio', 'social_media']
```

In [33]:
```
plt.bar(sales_columns, lasso_coef)
plt.xticks(rotation=45)
plt.show()
```



Expenditure on TV advertising is the most important feature in the dataset to predict sales values.

Chapter 3 - Fine-Tuning Your Model

Classification metrics

Accuracy: fraction of correctly classified samples

But not good for imbalanced classes (could just always predict majority class and couldn't identify e.g. fraudulent transactions)

Need to use confusion matrix (TN, FP, FN, TP)

In [2]:
```
from IPython.display import Image
Image(filename='images/cf.png', width=500)
```

Out[2]:

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Accuracy = $\frac{tp+tn}{tp+tn+fp+fn}$

Precision ('positive predictive value') = $\frac{tp}{tp+fp}$

- High precision = lower false positive rate (e.g. not many legitimate transactions are predicted to be fraudulent)
- Concerns all predicted fraudulent +ve class (e.g. all predicted to be fraudulent)
- Right column of confusion matrix

Recall ('sensitivity') = $\frac{tp}{tp+fn}$

- High recall = lower false negative rate (e.g. predicted most fraudulent transactions correctly)
- Concerns all actual +ve class (e.g. all actual fraudulent)

F1 score = $2 * \frac{precision*recall}{precision+recall}$

- Harmonic mean of precision and recall
- Gives equal weights to precision and recall
- Favours models with similar precision and recall

In [1]:
```
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import pandas as pd
```

In [2]:
```
diabetes_df = pd.read_csv('./datasets/diabetes_clean.csv')
```

In [3]:
```
diabetes_df_sub = diabetes_df[['bmi', 'age', 'diabetes']]
```

In [4]:
```
X_train, X_test, y_train, y_test = train_test_split(
    diabetes_df_sub.drop("diabetes", axis=1).values,
    diabetes_df_sub["diabetes"].values,
    test_size=0.3,
    random_state=42
)
```

In [5]:
```
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)
```

Out[5]:
```
KNeighborsClassifier(n_neighbors=6)
```

In [6]:
```
y_pred = knn.predict(X_test)
```

In [7]:
```
# Generate confusion matrix and classification report
print(confusion_matrix(y_test, y_pred)) # can add labels=[1,0] - careful of ordering!
print(classification_report(y_test, y_pred))
```

```
[[117  34]
 [ 47  33]]
              precision    recall  f1-score   support

           0       0.71      0.77      0.74       151
           1       0.49      0.41      0.45        80

    accuracy                           0.65       231
   macro avg       0.60      0.59      0.60       231
weighted avg       0.64      0.65      0.64       231
```

We see 117 TN, 34 FP, 47 FN and 33 TP. We see a better F1-score for the zero class (i.e. people without diabetes).

Precision (1): TP/(TP+FP) = 33/(33+34) = 0.49

Precision (0): TN/(TN+FN) = 117/(117+47) = 0.71

Recall (1): TP/(TP+FN) = 33/(33+47) = 0.41

Recall (0): TN/(TN+FP) = 117/(117+34) = 0.77

F1 (1): (2*(pr+re))/(pre+rec) = 0.45

F1 (0): (2*(pr+re))/(pre+rec) = 0.74

Accuracy: (TP+TN)/(TP+TN+FP+FN) = (33+117)/(33+117+34+47) = 0.65

Logistic regression and the ROC curve

Logistic regression:

- Used for classification problems
- Outputs probabilities:
  - If $p > 0.5$: data labeled 1

    - If $p < 0.5$: data labeled 0
- Produces a linear decision boundary

In [8]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
import matplotlib.pyplot as plt
```

In [9]:
```
X = diabetes_df.drop('diabetes', axis=1).values
y = diabetes_df['diabetes'].values
```

In [10]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42)
```

In [11]:
```
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

```
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[11]:
```
▾ LogisticRegression

LogisticRegression()
```

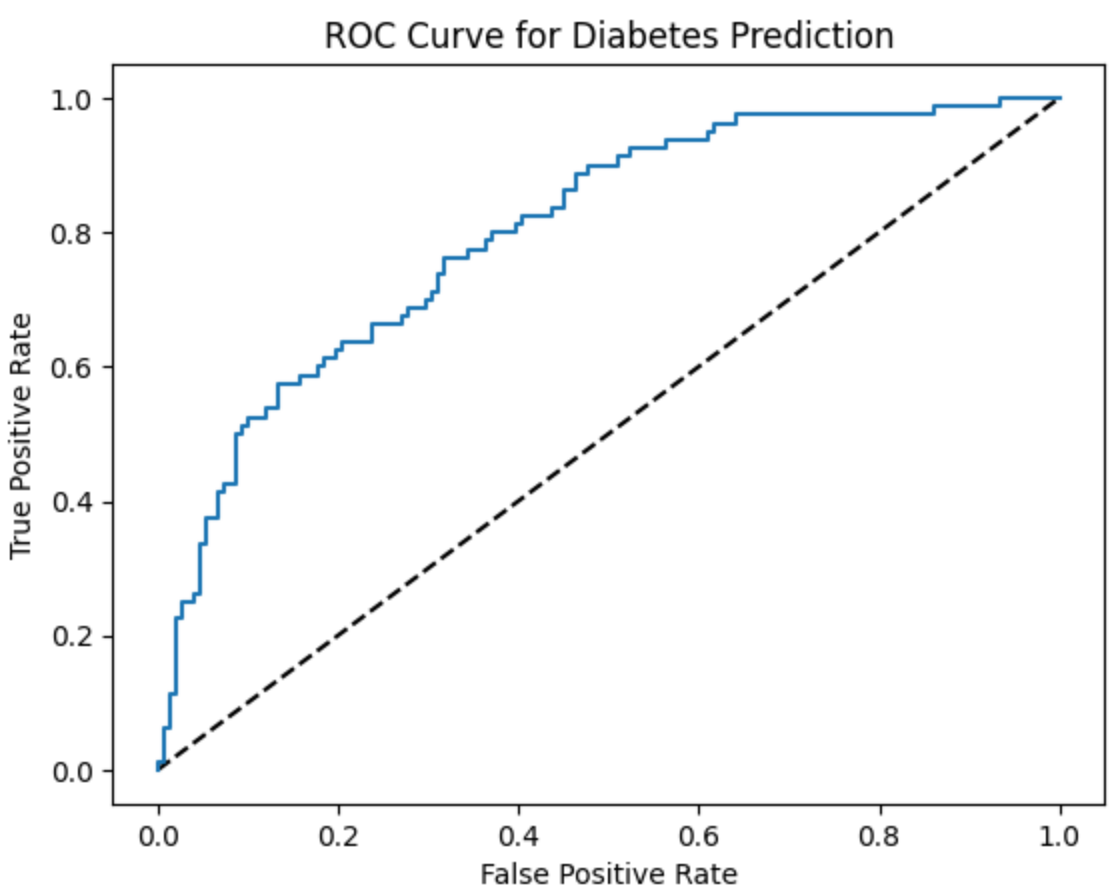
In [12]:
```
y_pred_probs = logreg.predict_proba(X_test)[:, 1]
y_pred_probs[:10]
```

Out[12]:
```
array([0.26551021, 0.18336638, 0.1211966 , 0.15613521, 0.4961118 ,
       0.4458219 , 0.01359249, 0.61646093, 0.55640529, 0.79311776])
```

Probability of a diabetes diagnosis for the first 10 individuals ranges from 0.01 to 0.79.

In [13]:
```
# Generate ROC curve values: fpr, tpr, thresholds
fpr, tpr, thresholds = roc_curve(y_test, y_pred_probs)
```

In [14]:
```
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Diabetes Prediction')
plt.show()
```



The model is much better than randomly guessing the class of each observation (ROC curve is above the dotted line, where the dotted line is a chance model - randomly guesses labels).

By default, logistic regression has a 0.5 threshold (can vary the threshold).

TPR = Recall = Sensitivity

FPR = 1 - Specificity = $1 - \frac{FP}{FP+TN}$

- If threshold=0: model predicts 1 for all observations, so predicts all +ve values correctly, and incorrectly predicts all -ve values (top right of ROC curve)
- If threshold=1: model predicts 0 for all data, so TPR and FPR = 0 (bottom left of ROC curve)
- If threshold=0.5: model above chance model line

Line is smoothed over different thresholds, which give different TPR and FPR values

ROC AUC: area under the curve (from 0-1, where 1 is ideal)

Perfect model: TPR=1, FPR=0

In [15]:
```
print(roc_auc_score(y_test, y_pred_probs))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.8002483443708608
[[117  34]
 [ 47  33]]
              precision    recall  f1-score   support

           0       0.71      0.77      0.74       151
           1       0.49      0.41      0.45        80

    accuracy                           0.65       231
   macro avg       0.60      0.59      0.60       231
weighted avg       0.64      0.65      0.64       231
```

ROC AUC score of 0.8002 means this model is 0.8002/0.5=60% better than a chance model at correctly predicting labels.

Hyperparameter tuning

Hyperparameters: parameters we specify before fitting the model (e.g. alpha and n_neighbors)

We try different hyperparameter values, fit all of them separately, see how well they perform, and choose the best performing values.

It's essential to use cross-validation to avoid overfitting to the test set (here we split the data and perform CV on the training set, and leave the test set for final evaluation).

In [16]:
```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold
import numpy as np
```

In [17]:
```
param_grid = {"alpha": np.linspace(0.00001, 1, 20)}
```

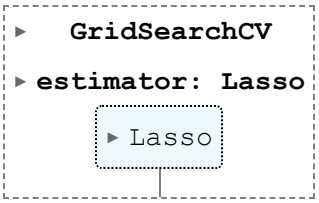
In [18]:
```
lasso = Lasso()
```

In [19]:
```
kf = KFold()
```

In [20]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

In [21]:
```
lasso_cv = GridSearchCV(lasso, param_grid, cv=kf)
lasso_cv.fit(X_train, y_train)
```

Out[21]:



In [22]:
```
print("Tuned lasso paramaters: {}".format(lasso_cv.best_params_))
print("Tuned lasso score: {}".format(lasso_cv.best_score_))
```

```
Tuned lasso paramaters: {'alpha': 1e-05}
Tuned lasso score: 0.27122338337314245
```

We see the best model only has an R-squared score of 0.27 (using the optimal hyperparameters does

not guarantee a high performing model).

Limitations of grid search: 10-fold CV, 3 hyperparameters, 30 total values = 900 fits! (can be computationally expensive)

In [23]:
```
params = {"penalty": ['l1', "l2"],
          "tol": np.linspace(0.0001, 1.0, 50),
          "C": np.linspace(0.1, 1.0, 50),
          "class_weight": ["balanced", {0: 0.8, 1: 0.2}]}
```

In [24]:
```
logreg = LogisticRegression()
```

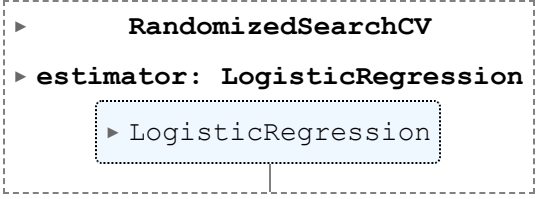
In [25]:
```
logreg_cv = RandomizedSearchCV(logreg, params, cv=kf)
logreg_cv.fit(X_train, y_train)
```

```
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
```

```
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
```

```
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/model_selection/_validation.py:378: FitFailedWarning:
35 fits failed out of a total of 50.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_score='ra
ise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
35 fits failed with the following error:
Traceback (most recent call last):
  File "/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/
python3.9/site-packages/sklearn/model_selection/_validation.py", line 686, in _fit_and_s
core
    estimator.fit(X_train, y_train, **fit_params)
  File "/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/
python3.9/site-packages/sklearn/linear_model/_logistic.py", line 1091, in fit
    solver = _check_solver(self.solver, self.penalty, self.dual)
  File "/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/
python3.9/site-packages/sklearn/linear_model/_logistic.py", line 61, in _check_solver
    raise ValueError(
ValueError: Solver lbfgs supports only 'l2' or 'none' penalties, got l1 penalty.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/model_selection/_search.py:953: UserWarning: One or more of the
test scores are non-finite: [       nan        nan        nan        nan        nan 0.75
077969
        nan 0.70522458        nan 0.75243236]
  warnings.warn(
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[25]:



In [26]:
```
print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Tuned Logistic Regression Best Accuracy Score: {}".format(logreg_cv.best_score_))
```

```
Tuned Logistic Regression Parameters: {'tol': 0.9387816326530612, 'penalty': 'l2', 'clas
s_weight': 'balanced', 'C': 0.9081632653061225}
Tuned Logistic Regression Best Accuracy Score: 0.7524323603891777
```

In [27]:
```
test_score = logreg_cv.score(X_test, y_test)
test_score
```

Out[27]:
```
0.6948051948051948
```

After trying a few hyperparameters, we get a model with 70% accuracy!

In [ ]:

Chapter 4 - Preprocessing and Pipelines

Creating dummy variables

In [1]:
```
import pandas as pd
```

In [2]:
```
music_df = pd.read_csv('./datasets/music_clean_orig.csv', index_col=[0])
```

In [3]:
```
music_df
```

Out[3]:

	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	s
0	41.0	0.644000	0.823	236533.0	0.814	0.687000	0.1170	-5.611	
1	62.0	0.085500	0.686	154373.0	0.670	0.000000	0.1200	-7.626	
2	42.0	0.239000	0.669	217778.0	0.736	0.000169	0.5980	-3.223	
3	64.0	0.012500	0.522	245960.0	0.923	0.017000	0.0854	-4.560	
4	60.0	0.121000	0.780	229400.0	0.467	0.000134	0.3140	-6.645	
...	...	...	...	...	...	...	...	...	
995	65.0	0.000983	0.531	216067.0	0.855	0.000000	0.0716	-4.950	
996	38.0	0.033200	0.608	218624.0	0.938	0.000000	0.3100	-2.681	
997	56.0	0.005790	0.939	144453.0	0.373	0.000000	0.2740	-7.779	
998	64.0	0.250000	0.546	178147.0	0.631	0.000000	0.1230	-5.757	
999	61.0	0.072500	0.641	-1.0	0.792	0.513000	0.1750	-6.453	

1000 rows × 12 columns

We need to build binary features for each song's genre.

In [4]:
```
# Convert categorical columns to series of dummy variables
# drop_first: produce k-1 dummy variables from original k
music_dummies = pd.get_dummies(music_df, drop_first=True)
```

In [5]:
```
music_dummies.shape
```

Out[5]:
```
(1000, 20)
```

9 new columns were added, as 10 values were in the 'genre' column. The 'genre' column was also dropped automatically.

Now, we will build a ridge regression model to predict song popularity.

In [6]:
```
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score, KFold
import numpy as np
```

In [7]:
```
X = music_dummies.drop(['popularity'], axis=1)
y = music_dummies['popularity']
```

In [8]:
```
ridge = Ridge(alpha=0.2)
```

In [9]:
```
kf = KFold(n_splits=5)
```

In [10]:
```
scores = cross_val_score(ridge, X, y, cv=kf, scoring="neg_mean_squared_error")
```

In [11]:
```
rmse = np.sqrt(-scores)
print("Average RMSE: {}".format(np.mean(rmse)))
print("Standard Deviation of the target array: {}".format(np.std(y)))
```

```
Average RMSE: 8.276818395409242
Standard Deviation of the target array: 14.02156909907019
```

Here we see the average RMSE of 8.27 is lower than the standard deviation of the target variable (song popularity), suggesting the model is reasonably accurate.

Missing values

In [12]:
```
music_df = pd.read_csv('./datasets/music_clean_missing.csv', index_col=[0])
```

In [13]:
```
print(music_df.isna().sum().sort_values())
```

```
genre                8
popularity          31
loudness            44
liveness            46
tempo               46
speechiness         59
duration_ms         91
instrumentalness    91
danceability       143
valence            143
acousticness       200
energy             200
dtype: int64
```

In [14]:
```
# Remove values where less than 5% are missing
music_df = music_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo
```

In [15]:
```
# Convert genre to a binary feature
music_df["genre"] = np.where(music_df["genre"] == "Rock", 1, 0)
```

In [16]:
```
print(music_df.isna().sum().sort_values())
print("Shape of the `music_df`: {}".format(music_df.shape))
```

```
popularity           0
liveness             0
loudness             0
tempo                0
genre                0
duration_ms         29
instrumentalness    29
speechiness         53
danceability       127
valence            127
acousticness       178
energy             178
dtype: int64
Shape of the `music_df`: (892, 12)
```

We have gone from 1000 observations to 892 after removing column data with less than 5% missing.

Pipelines

In [17]:
```
# Not sure what DataCamp is doing to the data between exercises :O
music_df = pd.read_csv('./datasets/music_clean_pipeline.csv', index_col=[0])
```

In [18]:
```
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, plot_confusion_matrix, classification_repo
import matplotlib.pyplot as plt
```

In [19]:
```
imputer = SimpleImputer() # defaults to mean imputation
```

In [20]:
```
knn = KNeighborsClassifier(n_neighbors=3)
```

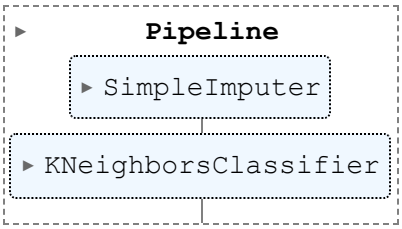
In [21]:
```
steps = [("imputer", imputer),
         ("knn", knn)]
```

In [22]:
```
X = music_df.drop(['genre'], axis=1).values
y = music_df['genre'].values
```

In [23]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
```

In [24]:
```
pipeline = Pipeline(steps)
pipeline.fit(X_train, y_train)
```

Out[24]:



In [25]:
```
y_pred = pipeline.predict(X_test)
```

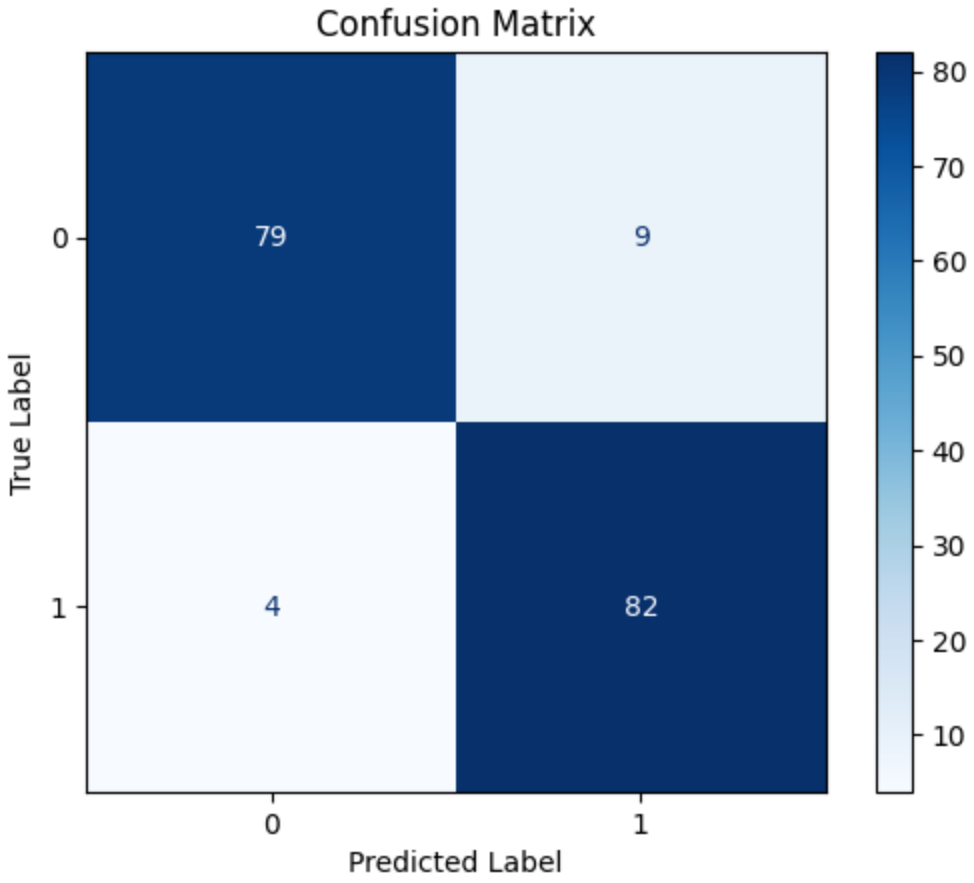
In [26]:
```
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[79  9]
 [ 4 82]]
              precision    recall  f1-score   support

           0       0.95      0.90      0.92        88
           1       0.90      0.95      0.93        86

    accuracy                           0.93       174
   macro avg       0.93      0.93      0.93       174
weighted avg       0.93      0.93      0.93       174
```

In [27]:
```
color = 'black'
matrix = plot_confusion_matrix(pipeline, X_test, y_test, cmap=plt.cm.Blues)
matrix.ax_.set_title('Confusion Matrix', color=color)
```

```
plt.xlabel('Predicted Label', color=color)
plt.ylabel('True Label', color=color)
plt.gcf().axes[0].tick_params(colors=color)
plt.gcf().axes[1].tick_params(colors=color)
plt.show()
```

```
/Users/harrybaines/Documents/Coding/DataCamp-ML-Scientist-Track/datacampenv/lib/python3.
9/site-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function plot_confusion_
matrix is deprecated; Function `plot_confusion_matrix` is deprecated in 1.0 and will be
removed in 1.2. Use one of the class methods: ConfusionMatrixDisplay.from_predictions or
ConfusionMatrixDisplay.from_estimator.
  warnings.warn(msg, category=FutureWarning)
```



Centering and scaling

Here we will build a pipeline to preprocess the music dataset features and build a lasso regression model to predict a song's loudness.

In [28]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Lasso
```

In [29]:
```
music_df = pd.read_csv('./datasets/music_clean_center_scaling.csv', index_col=[0])
```

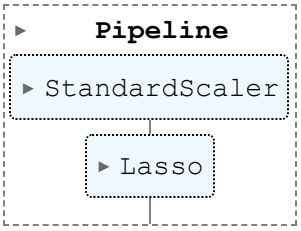
In [30]:
```
X = music_df.drop(['loudness'], axis=1).values
y = music_df['loudness'].values
```

In [31]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=5)
```

In [32]:
```
steps = [("scaler", StandardScaler()),
         ("lasso", Lasso(alpha=0.5))]
```

In [33]:
```
pipeline = Pipeline(steps)
pipeline.fit(X_train, y_train)
```

Out[33]:



In [34]:
```
print(pipeline.score(X_test, y_test))
```

```
0.6294826383621106
```

We get an R-squared of 0.629, compared to an R-squared of 0.393 without scaling!

Next, we will build a peipline to scale features in `music_df` and use grid search CV using a logistic regression model with different C values, to predict the target 'genre'.

In [35]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
```

In [36]:
```
music_df = pd.read_csv('./datasets/music_clean_center_scaling.csv', index_col=[0])
```

In [37]:
```
steps = [("scaler", StandardScaler()),
         ("logreg", LogisticRegression())]
```

In [38]:
```
X = music_df.drop(['genre'], axis=1).values
y = music_df['genre'].values
```

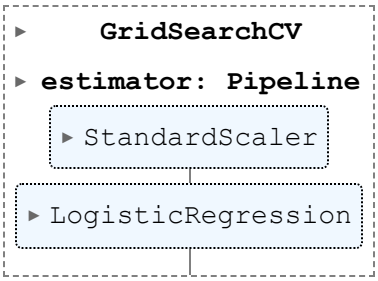
In [39]:
```
pipeline = Pipeline(steps)
```

In [40]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=21)
```

In [41]:
```
parameters = {"logreg__C": np.linspace(0.001, 1.0, 20)}
```

In [42]:
```
cv = GridSearchCV(pipeline, param_grid=parameters)
cv.fit(X_train, y_train)
```

Out[42]:



In [43]:
```
print(cv.best_score_, "\n", cv.best_params_)
```

```
0.8425
 {'logreg__C': 0.1061578947368421}
```

We get a final model with an accuracy of 0.8425 with a C value of approximately 0.1.

Finally, we will build three regression models to predict a song's 'energy' levels.

In [44]:
```
from sklearn.linear_model import LinearRegression
```

In [45]:
```
music_df = pd.read_csv('./datasets/music_clean_missing.csv', index_col=[0])
```

In [46]:
```
music_df = music_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo
```

In [47]:
```
music_dummies = pd.get_dummies(music_df, drop_first=True)
```

In [48]:
```
models = {
    "Linear Regression": LinearRegression(),
    "Ridge": Ridge(alpha=0.1),
    "Lasso": Lasso(alpha=0.1)
}
```

In [49]:
```
X = music_dummies.drop(['energy'], axis=1).values
y = music_dummies['energy'].values
```

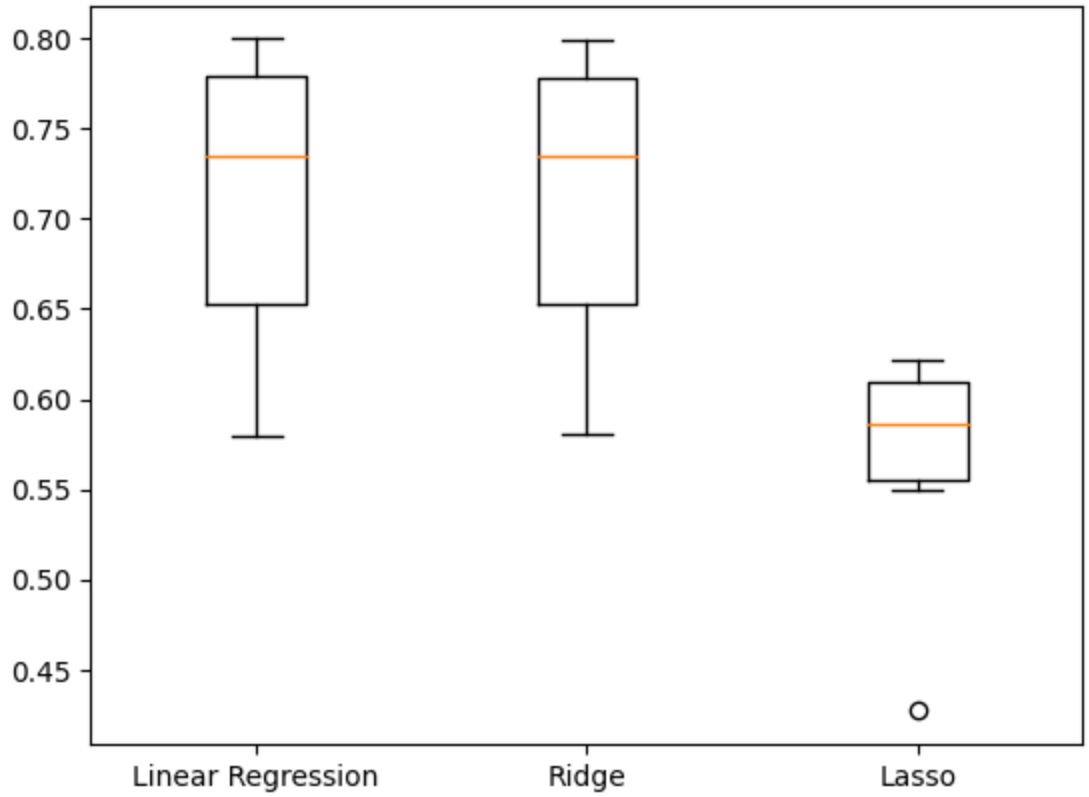
In [50]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=21)
```

In [51]:
```
imputer = SimpleImputer()
```

In [52]:
```
X_train = imputer.fit_transform(X_train)
y_train = imputer.fit_transform(y_train.reshape(-1,1))
X_test = imputer.fit_transform(X_test)
y_test = imputer.fit_transform(y_test.reshape(-1,1))
```

In [53]:
```
results = []

# Loop through the models' values
for model in models.values():
    kf = KFold(n_splits=6, random_state=42, shuffle=True)

    # Perform cross-validation
    cv_scores = cross_val_score(model, X_train, y_train, cv=kf)

    # Append the results
    results.append(cv_scores)
```

In [54]:
```
# Create a box plot of the results
plt.boxplot(results, labels=models.keys())
plt.show()
```



We see lasso regression is not a good model for this problem, while linear regression and ridge regression perform fairly equally.

Next, we will check predictive performance on the test set to see if either linear or ridge regression are better. We use RMSE as the metric.

In [55]:
```
from sklearn.metrics import mean_squared_error
```

In [56]:
```
scaler = StandardScaler()
```

In [57]:
```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.fit_transform(X_test)
```

In [58]:
```
for name, model in models.items():

    # Fit the model to the training data
    model.fit(X_train_scaled, y_train)

    # Make predictions on the test set
    y_pred = model.predict(X_test_scaled)

    # Calculate the test_rmse
    test_rmse = mean_squared_error(y_test, y_pred, squared=False)
    print("{} Test Set RMSE: {}".format(name, test_rmse))
```

```
Linear Regression Test Set RMSE: 0.12267024610658882
Ridge Test Set RMSE: 0.12267397733287912
Lasso Test Set RMSE: 0.17292833360218704
```

The linear regression model only slightly edges the best performance.

Next, let's build a model to classify whether a song is popular or not.

In [59]:
```
from sklearn.tree import DecisionTreeClassifier
```

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

In [60]:
```
music_dummies['popularity'] = np.where(
    music_dummies['popularity'] > music_dummies['popularity'].median(),
    1,
    0
)
```

In [61]:
```
X = music_dummies.drop(['popularity'], axis=1).values
y = music_dummies['popularity'].values
```

In [62]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=21)
```

In [63]:
```
imputer = SimpleImputer()
```

In [64]:
```
X_train = imputer.fit_transform(X_train)
y_train = imputer.fit_transform(y_train.reshape(-1, 1)).ravel()
X_test = imputer.fit_transform(X_test)
y_test = imputer.fit_transform(y_test.reshape(-1, 1)).ravel()
```

In [65]:
```
scaler = StandardScaler()
```

In [66]:
```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.fit_transform(X_test)
```

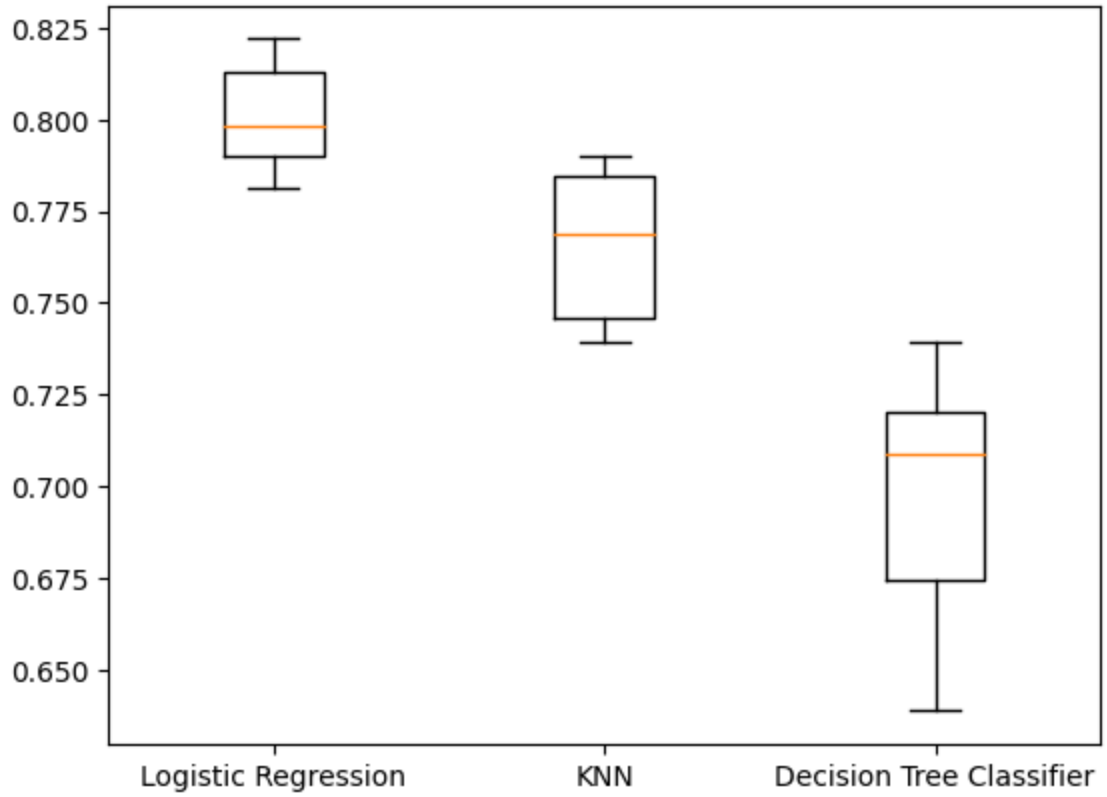
In [67]:
```
X_train_scaled.shape
```

Out[67]:
```
(713, 19)
```

In [68]:
```
models = {
  "Logistic Regression": LogisticRegression(),
  "KNN": KNeighborsClassifier(),
  "Decision Tree Classifier": DecisionTreeClassifier()
}
```

In [69]:
```
results = []

# Loop through the models' values
for model in models.values():

    # Instantiate a KFold object
    kf = KFold(n_splits=6, random_state=12, shuffle=True)

    # Perform cross-validation
    cv_results = cross_val_score(model, X_train_scaled, y_train, cv=kf)
    results.append(cv_results)
```

In [70]:
```
plt.boxplot(results, labels=models.keys())
plt.show()
```



Logistic regression seems to be the best model based on the cross-validation results.

Finally, we will build a pipeline to perform all preprocessing operations and perform hyperparameter tuning of a logistic regression model. We will find the best parameters and accuracy when predicting song genre.

In [71]:
```
music_df = pd.read_csv('./datasets/music_clean_center_scaling.csv', index_col=[0])
```

In [72]:
```
X = music_df.drop(['genre'], axis=1).values
y = music_df['genre'].values
```

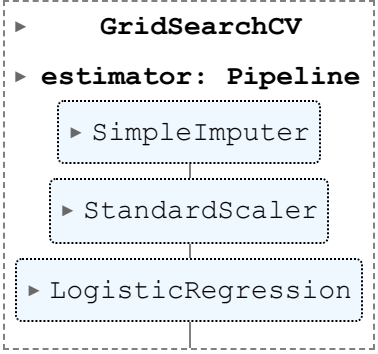
In [73]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=21)
```

In [74]:
```
steps = [("imp_mean", SimpleImputer()),
         ("scaler", StandardScaler()),
         ("logreg", LogisticRegression())]
```

In [75]:
```
pipeline = Pipeline(steps)
params = {"logreg__solver": ["newton-cg", "saga", "lbfgs"],
         "logreg__C": np.linspace(0.001, 1.0, 10)}
```

In [76]:
```
tuning = GridSearchCV(pipeline, param_grid=params)
tuning.fit(X_train, y_train)
```

Out[76]:



In [77]:
```
y_pred = tuning.predict(X_test)
```

In [78]:
```
print("Tuned Logistic Regression Parameters: {}, Accuracy: {}".format(tuning.best_params
```

```
Tuned Logistic Regression Parameters: {'logreg__C': 0.112, 'logreg__solver': 'newton-c
g'}, Accuracy: 0.825
```

We get a final model which is 82% accuracy in predicting song genres.

In [ ]: