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


from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import auc
from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import LogisticRegression, Ridge


from matplotlib.collections import EllipseCollection
from matplotlib.colors import Normalize
```

Outline

1. Data preprocessing

We will use the Car price dataset. Download it from

`wget https://raw.githubusercontent.com/alexeygrigorev/mlbookcamp-code/master/chapter-02-car-price/data.csv`
We'll work with the `MSRP` variable, and transform it to a classification task.

We need to use only these features:

- `Make`,
- `Model`,
- `Year`,
- `Engine HP`,
- `Engine Cylinders`,
- `Transmission Type`,
- `Vehicle Style`,
- `highway MPG`,
- `city mpg`,
- `MSRP`

For data preparation, we need to do the following:

- Keep only the columns above
- Lowercase the column names and replace spaces with underscores
- Fill the missing values with 0
- Make the price binary (1 if above the average, 0 otherwise) - this will be our target variable above_avg
- Split the data into 3 parts: train/validation/test with 60%/20%/20% distribution. Use train_test_split function for that with random_state=1

In [ ]:
```python
df = pd.read_csv('data/car_price.csv')
```

In [ ]:
```python
# Feature selection
features = ['Make', 'Model', 'Year', 'Engine HP',
            'Engine Cylinders', 'Transmission Type', 'Vehicle Style',
            'highway MPG', 'city mpg', 'MSRP']

df = df[features]

# lowercase for columns names and replace spaces
df.columns = df.columns.str.lower()\
                        .str.replace(' ','_')

# Missing values
# Rename target column and fill missing values with zero
df.rename(columns= {'msrp':'price'}, inplace = True)
df[['engine_hp', 'engine_cylinders']] = df[['engine_hp', 'engine_cylinders']].fillna(0)

# Making numerical variables binary
```

```
avg_price = df['price'].mean()
above_avg = df['price'] > avg_price

# Binary variables: if above average
#1: True
#0: False
df['above_avg'] = above_avg.astype(int)

display(df[['price', 'above_avg']].head())
print('Average:', round(avg_price, 0))

df.head(2)

# delete price
del df['price']
```

	price	above_avg
0	46135	1
1	40650	1
2	36350	0
3	29450	0
4	34500	0

```
Average: 40595.0
```

In [ ]:
```
# Split data : 100% -> 1*(1 - 0.2) = 0.8/ 1*0.2 = 0.2
df_full_train, df_test = train_test_split(df, test_size = 0.2,  random_state=1)
# Split again : 80% -> 0.8(1 - 0.25) = 0.6/ 0.8*0.25 = 0.2
df_train, df_val = train_test_split(df_full_train, test_size = 0.25,  random_state=1)

# save target values
Y_train = df_train['above_avg'].values
Y_val = df_val['above_avg'].values

# take out the target values from the dataframe
del df_train['above_avg']
del df_val['above_avg']
```

2. Confusion Matrix for Binary classes

Consider a hypothetical dataset with multiple features $\mathbf{X}_1, \cdots, \mathbf{X}_d$ and a target variable $\mathbf{Y}$ as shown:

$$\left(\begin{array}{c|ccc|c} \text{Instance} & \mathbf{X}_1 & \cdots & \mathbf{X}_d & \mathbf{Y} \\ \hline \mathbf{x}_1 & x_{11} & \cdots & x_{1d} & y_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{x}_n & x_{n1} & \cdots & x_{nd} & y_n \end{array}\right).$$

Each row vector is an instance $\mathbf{x}_i = (x_{i1}, \ldots, x_{id})$ with d features. The dataset can be represented as a feature matrix $\mathbf{X}$ and a target vector $\mathbf{Y}$:

$$\mathbf{X} = \begin{pmatrix} x_{11} & \cdots & x_{1d} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nd} \end{pmatrix} \quad \text{and} \quad \mathbf{Y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

The target vector $\mathbf{Y}$ is binary. Each element y_i in $\mathbf{Y}$ corresponds to one of two possible class labels, which is defined by the set $\mathcal{Y} = \{c_-, c_+\} = \{0, 1\}$. Based in these unique classes in $\mathcal{Y}$ we can categorize or partition instances $\mathbf{x}_i$ **based on their true class labels** into groups c_+ and c_-. This leads to the dataset partitioned by its true labels denoted as $\mathcal{D} = \{\mathbf{D}_-, \mathbf{D}_+\}$.

For each instance $\mathbf{x}_i$, if it belongs to the true class $y_i = c_-$, it is expressed in set notation as:

$$\mathbf{D}_- = \{\mathbf{x}_i | y_i = c_-\}$$

Where $n_- = |\mathbf{D}_-|$ denotes the number of true class instances for class $y_i = c_-$. Similarly, for the positive class, we would have $n_+ = |\mathbf{D}_+|$

While $\mathbf{D}$ represents the true labels, the model's predictions class labels are also crucial for comparing. Let's use $\mathcal{R} = \{\mathbf{R}_-, \mathbf{R}_+\}$ to denote partitions **based on predicted values** $\hat{\mathbf{Y}}$. So, for each instance $\mathbf{x}_i$, if it belongs to the predicted class $\hat{y} = c_-$, it is expressed as:

$$\mathbf{R}_- = \{\mathbf{x}_i | \hat{y}_i = c_-\}$$

Here $m_- = |\mathbf{R}_-|$ denotes the number of instances the model predicts to belong to class c_-. Similarly, $m_+ = |\mathbf{R}_+|$ for the positive class.

Taking into account both the number of true and predicted labels for each class, we can form a 2×2 matrix. This matrix, represent the intersections of each partitions based on some class from $\mathcal{Y}$ and is know as confusion matrix.

$$\mathbf{N} = \left(\begin{array}{c|cc} y_i | \ \hat{y}_i & c_- & c_+ \\ \hline c_- & n_{00} & n_{01} \\ c_+ & n_{10} & n_{11} \end{array} \right).$$

Each entry of the matrix denotes the size of the intersection $n_{ij} = |\mathbf{R}_i \cap \mathbf{D}_j|$ for $i, j = \{-, +\} = \{0, 1\}$. In the context of binary classification, each element of the matrix is given a specific name:

- **True Negatives (TN)**: the number of instances that the classifier correctly predicts as negative:

$$\mathrm{TN} = n_{00} = |\mathbf{R}_- \cap \mathbf{D}_-| = |\{\mathbf{x}_i | \hat{y}_i = y_i = c_-\}|$$

- **False Negatives (FN)**: the number of instances that the classifier predicts to be in the negative class, which in fact belongs to the positive class:

$$\mathrm{FN} = n_{10} = |\mathbf{R}_- \cap \mathbf{D}_+| = |\{\mathbf{x}_i | \hat{y}_i = c_- \ \text{ and } \ y_i = c_+\}|$$

- **False Positives (FP)**:the number of instances that the classifier predicts to be positive, which in fact belong to the negative class:

$$\mathrm{FP} = n_{01} = |\mathbf{R}_+ \cap \mathbf{D}_-| = |\{\mathbf{x}_i | \hat{y}_i = c_+ \ \text{ and } \ y_i = c_-\}|$$

- **True Positives (TP)**: the number of instances that the classifier correctly predicts as positive:

$$\mathrm{TP} = n_{11} = |\mathbf{R}_+ \cap \mathbf{D}_+| = |\{\mathbf{x}_i | \hat{y}_i = y_i = c_+\}|$$

As an example, let's consider a dataset of cars and their price relative to the average price. Here, the class $c_- = 0$ represents a price **below average** and $c_+ = 1$ indicates a price **above average**. Using a car price dataset, we can distribute all cars (instances) across the categories defined above.

No description has been provided for this image

- **True Positives (TP)**: Cars with prices **above average** correctly predicted as **above average**.
- **True Negatives (TN)**: Cars with prices **below average** correctly predicted as **below average**.
- **False Positives (FP)**: Cars with prices **below average** incorrectly predicted as **above average**.
- **False Negatives (FN)**: Cars with prices **above average** incorrectly predicted as **below average**.

Metrics

To understand the performance of any classification model, it's crucial to look into metrics that evaluate how well it has classified the data. One way to visually inspect these results is through a confusion matrix.

Consider the confusion matrix presented as:

$$\mathbf{N} = \left(\begin{array}{c|cc} y_i | \ \hat{y}_i & c_- & c_+ \\ \hline c_- & n_{00} & n_{01} \\ c_+ & n_{10} & n_{11} \end{array} \right).$$

As explained before, each element of this matrix are labeled as $\mathrm{TN} = n_{00}$ for the True Negative, $\mathrm{FN} = n_{10}$ for the False Negative, $\mathrm{FP} = n_{01}$ for the False Positive, and $\mathrm{TP} = n_{11}$ for the True Positive. With this matrix in hand, we can proceed to extract the metrics that quantifies the model's performance.

- **Global Measures**

  These are measures that evaluate the overall performance of the classifier across all classes. They provide a way of quantifying how the model is behaving.

  - **Error rate**: ratio of all the misclassified instances to the total number of instances n in the dataset. Quantifies the proportion of instances that the classifier predicted incorrectly

$$\text{Error Rate} = \frac{\mathrm{FP} + \mathrm{FN}}{n} = \frac{FP + FN}{FP + FN + TP + TN}$$

  - **Accuracy**: ratio of all correctly classified instances to the total number of instances n in the dataset. In simpler terms, it quantifies the proportion of instances that the classifier got right.

$$\text{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{n}$$

    However, relying on accuracy alone can sometimes be misleading, especially in imbalanced datasets where one class is predominant.

While global measures provide a comprehensive way of quantifying of the classifier's performance, they might not always reflect the efficiency of the classifier's model in specific scenarios or with respect to individual classes.

- **Class Specific Measures**

  For a more nuanced view of performance, especially when dealing with datasets where some classes are more important or more challenging to classify than others, class-specific measures can be used instead of global measures. These measures quantify the model's performance by their individual classes, helping identify potential areas of improvement in the model.

  - **Precision**: The ratio of **correctly classified instances** of a specific class to **all instances of that class**. Essentially, it quantifies how trustworthy the classifier's predictions are for a specific class.

    For the negative class:

    $$\text{prec}_{\_} = \frac{n_{00}}{m_{\_}} = \frac{\text{TN}}{\text{TN} + \text{FN}}$$

    which quantifies how many of the instances predicted as negative truly are negative. For the positive class:

    $$\text{prec}_{+} = \frac{n_{11}}{m_{+}} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

    where $m_{+} = |\mathbf{R}_{+}|$ and $m_{\_} = |\mathbf{R}_{\_}|$ are the size of the partitions for predicted classes c_{+} and $c_{\_}$, respectively.

  - **Recall**: The ratio of **correctly classified instances** of a specific class to **all true instances of that class**. Essentially, it quantifies the model's ability to identify and correctly classify all relevant instances.

    - **True Negative Rate (Specificity)**: The proportion of true negative instances that were correctly identified and classified. Is the efficiency in correctly predicting the negative class when it is indeed negative.

      $$\text{recall}_{\_} = \text{Specificity} = \frac{n_{00}}{n_{\_}} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

      where $n_{\_} = |\mathbf{D}_{\_}| = \text{TN} + \text{FP}$ is the size of the partition for true negative class.

    - **True Positive Rate (Sensitivity)**: the proportion of true positive instances that were correctly identified and classified. Is the efficiency in correctly predicting the positive class when it is indeed positive.

      $$\text{recall}_{+} = \text{Sensitivity} = \frac{n_{11}}{n_{+}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

      where $n_{+} = |\mathbf{D}_{+}| = \text{TP} + \text{FN}$ is the size of the partition for true positive class.

  - **Miss Rate ($1-$ Recall)**: The ratio of **incorrectly classified instances** of a specific class to **all true instances of that class**. Essentially, it quantifies the model's errors or limitations in classifying instances correctly.

    - **False Negative Rate (FNR)**: The proportion of true positive instances that the model incorrectly classifies as negative. In essence, it quantifies the missed opportunities by the classifier for the positive class.

      $$1 - \text{Sensitivity} = \frac{n_{10}}{n_{+}} = \frac{\text{FN}}{\text{TP} + \text{FN}}$$

    - **False Positive Rate (FPR)**: The proportion of true negative instances that the model incorrectly classifies as positive. This metric gives insight into the model's over-predicting tendencies for the positive class when instances are true negative.

      $$1 - \text{Specificity} = \frac{n_{01}}{n_{\_}} = \frac{\text{FP}}{\text{TN} + \text{FP}}$$

  - **F1 Score**

    The F1 score is a metric that combines both precision and recall, providing a single score that balances the trade-off between the two for binary classification:

    - If the classifier predicts nearly every instance as positive, achieving a recall close to 1, its precision often decreases.
    - If the classifier is very selective and achieves a high precision by predicting only a few true instances as positive, its recall might be compromised due to missed true positives.

    The F1 score for binary classification is calculated as the harmonic mean of precision and recall:

    $$F1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

    For a perfect classifier, the F1 score is 1, indicating an optimal balance between precision and recall for both classes.

3. ROC Curve

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a binary classifier as the decision threshold is varied. Traditionally, we utilize a threshold of $t = 0.5$, but the ROC curve enables us to visualize the performance across all possible thresholds.

In a prediction model, $P(\hat{\mathbf{y}}_i = c_-|\mathbf{x}_i) < t$ is the predicted probability of an instance belonging to class c_-, whereas $P(\hat{\mathbf{y}}_i = c_+|\mathbf{x}_i) \geq t$ epresents the predicted probability of an instance belonging to class c_+. The threshold t is of particular importance, as varying it can lead to different trade-offs between **Sensitivity** and **Specificity**. For instance, in some applications, minimizing false positives is crucial, so a higher threshold that emphasizes precision would be chosen.

The ROC curve represents the True Positive Rate (TPR) or sensitivity (y-axis) against the False Positive Rate (FPR) or 1-specificity (x-axis) as the decision threshold varies. Each point on the ROC curve corresponds to a unique pair of TPR and FPR values associated with a specific decision threshold.

For each distinctive score, we plot a point with the coordinates:

$$(FPR, TPR) = (\frac{n_{01}}{n_-}, \frac{n_{11}}{n_+})$$

Remember that:

- **False Positive Rate (FPR)**: Ratio of true negative instances incorrectly identified as positive

$$\mathrm{FPR} = \frac{n_{01}}{n_-} = \frac{\mathrm{FP}}{\mathrm{TN} + \mathrm{FP}}$$

- **True Positive Rate (Sensitivity)**: Ratio of true positive instances correctly identified.

$$\mathrm{TPR} = \frac{n_{11}}{n_+} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

An ideal classifier's ROC curve will reach the plot's top left corner, implying a TPR of 1 and FPR of 0. Conversely, a classifier performing no better than random guessing will have its ROC curve align with the 45-degree diagonal of the plot.

3.1 AUC Score

The Area Under the Curve (AUC) is a metric that captures the performance of a binary classifier. The AUC represents the probability that a randomly selected positive instance is ranked (given a higher predicted probability) higher than a randomly selected negative instance, thus reflecting the model's ability to rank instances correctly.

In this scenario:

- If the classifier perfectly predicts (i.e., every prediction matches the true outcome), the ROC curve will stretch to the plot's top left corner, given an AUC score of 1. This condition also corresponds to an ideal situation where there's no overlap between the scores for the two classes, indicating perfect discrimination.
- If a classifier cannot discriminate between the classes at all (i.e., it identifies equal numbers of false positives and true positives across all thresholds), the ROC curve will form a 45-degree diagonal line on the plot, which means the AUC is 0.5, an indication of random guessing.

The AUC-ROC metric is advantageous as it provides a reliable measure of the model's performance, independent of the dataset's balance. Furthermore, the AUC emphasizes the model's ranking of predictions, assessing the quality of the probability scores it assigns rather than just their absolute values.

3.2 AUC for Feature Importance

While AUC is commonly used to assess the overall performance of a binary classifier, it can also be utilized as a tool for evaluating the predictive power of individual features. By treating each feature as a rudimentary "classifier", the AUC provides insights on the feature's capability to differentiate between classes

Instead of using the predicted scores or probabilities from a machine learning model, we use the actual numbers from each feature as the basis for ranking. The main idea of AUC stays consistent: it measures how well the feature, when viewed as a simple classifier, can separate the classes.

If a particular feature has an AUC score of less than 0.5, it suggests that the feature has a negative correlation with the target variable. By multiplying the feature values by -1, the relationship becomes positive, improving its rank ordering capability.

Using this technique, we can rank the features based on their AUC scores. Features with higher AUC values have stronger univariate predictive power.

3.3 Questions 1-4

- **Question 1: ROC/AUC feature importance**

ROC/AUC could also be used to evaluate feature importance of numerical variables.

- For each numerical variable, use it as score and compute AUC with the `above_average` variable
- Use the training dataset for that

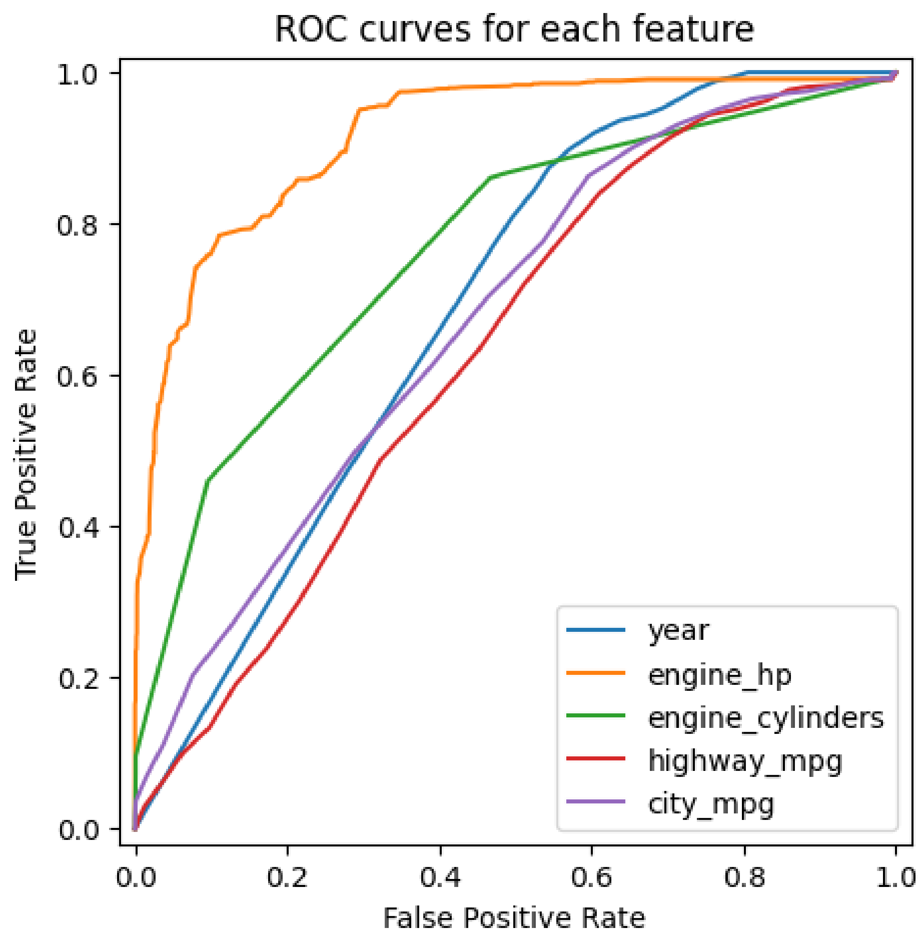
If AUC < 0.5, invert this variable by putting "-" in front (e.g. `-df_train['engine_hp']`). Which numerical variable has the highest AUC?

- **`engine_hp`**
- `engine_cylinders`
- `highway_mpg`
- `city_mpg`

In [ ]:
```
def roc_prec_recall(feature, target, thresholds = None, feature_selection = True):

    feature = np.array(feature)
    target = np.array(target)

    # Mergesort is a stable sort,
    # The order of equal elements will not change after sorting.
    sorted_indices = np.argsort(feature, kind="mergesort")[::-1]
    feature_sorted = feature[sorted_indices]
    target_sorted = target[sorted_indices]
    if thresholds is None:
        thresholds = np.unique(feature_sorted)


    tpr_list = [0]
    fpr_list = [0]
    precision_list = []
    recall_list = []
    # Predicted positives and negatives based on current threshold
    for threshold in thresholds:

        # Predict classes
        predicted_pos = feature >= threshold
        predicted_neg = feature < threshold

        # True classes
        true_pos = target == 1
        true_neg = target == 0


        tp = (true_pos & predicted_pos).sum()
        tn = (true_neg & predicted_neg).sum()
        fp = (predicted_pos & true_neg).sum()
        fn = (predicted_neg & true_pos).sum()


        tpr = tp/(fn + tp )
        fpr = fp/(fp + tn )
        precision = tp/(tp+fp)
        recall = tp/(tp + fn)

        tpr_list.append(tpr)
        fpr_list.append(fpr)
        precision_list.append(precision)
        recall_list.append(recall)

    tpr = sorted(tpr_list)
    fpr = sorted(fpr_list)
    precision = precision_list
    recall = recall_list

    AUC = auc(fpr, tpr)

    if AUC <= 0.5 and feature_selection:
        feature = -feature
        return roc_prec_recall(feature, target)

    return AUC, fpr, tpr, precision, recall
```

In [ ]:
```
# Numerical features
mask = df.dtypes != object
numerical = list(df_train.dtypes[mask].index)
```

```
plt.figure(figsize=(5, 5))

# For each numerical feature, compute its importance and plot its ROC curve.
auc_dic = {}
for feature in numerical:
    current_auc, fpr, tpr, _, _ = roc_prec_recall(df_train[feature], Y_train)
    auc_dic[feature] = round(current_auc,2)
    plt.plot(fpr, tpr, label=f'{feature}')

# Setting the plot legend, limits, labels, and title.
plt.legend()
plt.xlim([-0.02, 1.02])
plt.ylim([-0.02, 1.02])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curves for each feature')

plt.show()

# Convert the dictionary to a DataFrame
auc_df = pd.DataFrame(list(auc_dic.items()), columns=['Feature', 'AUC'])\
                                .sort_values(by='AUC', ascending=False)

display(auc_df)
```



	Feature	AUC
1	engine_hp	0.92
2	engine_cylinders	0.77
0	year	0.69
4	city_mpg	0.67
3	highway_mpg	0.63

- **Question 2: Training the model**

Apply one-hot-encoding using `DictVectorizer` and train the logistic regression with these parameters:

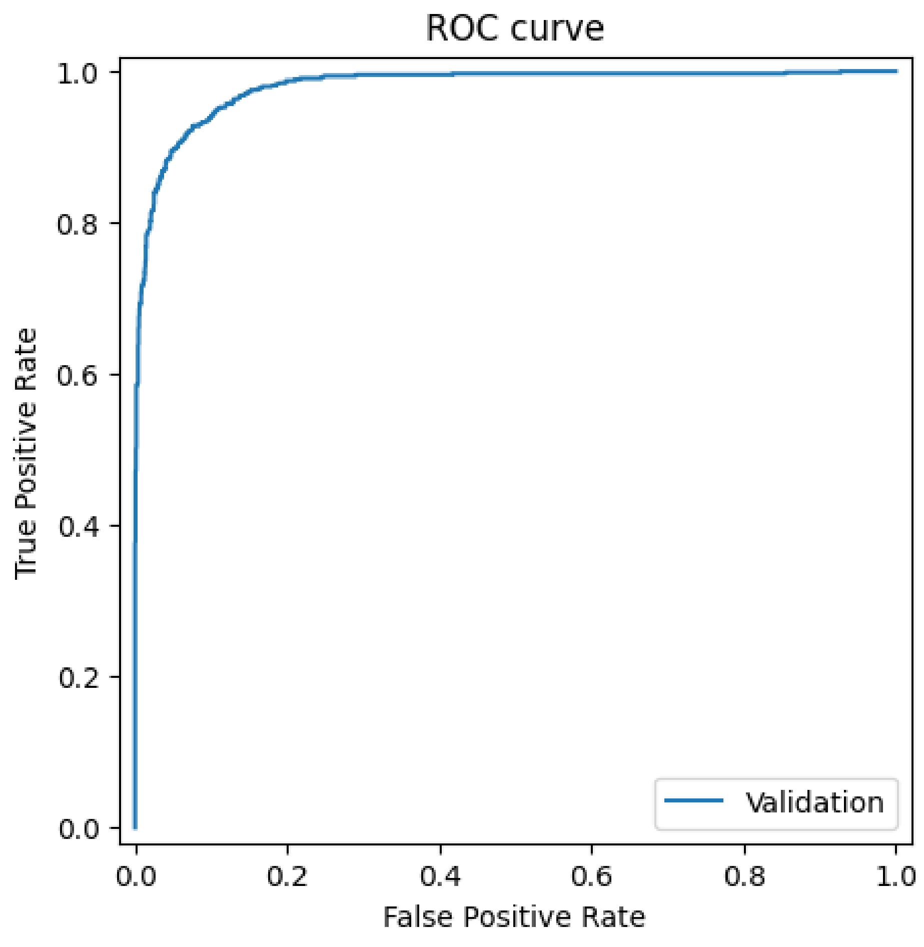
`LogisticRegression(solver='liblinear', C=1.0, max_iter=1000)`
What's the AUC of this model on the validation dataset? (round to 3 digits)

- 0.678
- 0.779
- 0.878
- **0.979**

```
In [ ]: train_dict = df_train.to_dict(orient = 'records')
        val_dict = df_val.to_dict(orient = 'records')

        # One-Hot encode
        dv = DictVectorizer(sparse = False)
        X_train = dv.fit_transform(train_dict)
```

```
X_val = dv.transform(val_dict)

model = LogisticRegression(solver='liblinear',C = 1.0, max_iter=1000)
model.fit(X_train, Y_train)

# get predicted probabilities for the positive class
Y_pred_proba = model.predict_proba(X_val)[:, 1]
```

In [ ]:
```
plt.figure(figsize=(5, 5))
auc_value, fpr,tpr, __,_ = roc_prec_recall(Y_pred_proba, Y_val, feature_selection=False)
plt.plot(fpr, tpr, label=f'Validation')

# Settings
plt.legend()
plt.xlim([-0.02, 1.02])
plt.ylim([-0.02, 1.02])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve')

plt.show()

print('AUC =', round(auc_value, 3) )
```



```
AUC = 0.98
```

- **Question 3: Precision and Recall**

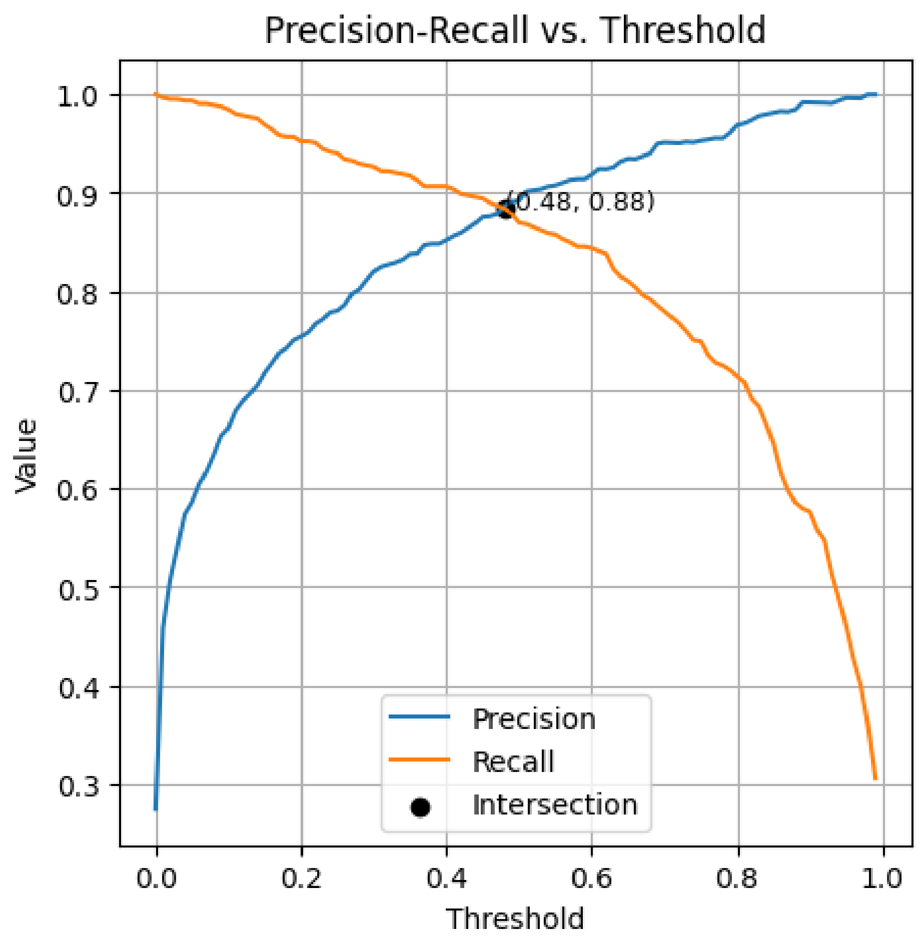
Now let's compute precision and recall for our model.

- Evaluate the model on all thresholds from 0.0 to 1.0 with step 0.01
- For each threshold, compute precision and recall
- Plot them

At which threshold precision and recall curves intersect?

- 0.28
- **0.48**
- 0.68
- 0.88

In [ ]:
```
# Compute precision and recall using your function
thresholds = np.arange(0.0, 1.0 ,0.01)
_, _,_, precision, recall = roc_prec_recall(Y_pred_proba, Y_val, thresholds, feature_selection = False)

# Intersection point between precision and recall
intersect_point = np.abs(np.array(precision) - np.array(recall))
idx = np.argmin(intersect_point)
intersect_threshold =  thresholds[idx]
intersect_recall = recall[idx]

plt.figure(figsize=(5, 5))
_, _,_, precision, recall = roc_prec_recall(Y_pred_proba, Y_val, thresholds, feature_selection = False)
plt.plot(thresholds, precision, label=f'Precision')
plt.plot(thresholds, recall, label=f'Recall')
plt.scatter(intersect_threshold,intersect_recall, color = 'black', label='Intersection')
plt.text(   intersect_threshold, intersect_recall,
            f'({intersect_threshold:.2f}, {intersect_recall:.2f})', color = 'black', fontsize=9)

# Settings
plt.legend()
plt.xlabel('Threshold')
plt.ylabel('Value')
plt.title('Precision-Recall vs. Threshold')
plt.grid(True)
plt.show()
```



- **Question 4: F1 score**

Precision and recall are conflicting - when one grows, the other goes down. That's why they are often combined into the F1 score - a metrics that takes into account both

This is the formula for computing F1:

$$F1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Where P is precision and R is recall.

Let's compute F1 for all thresholds from 0.0 to 1.0 with increment 0.01

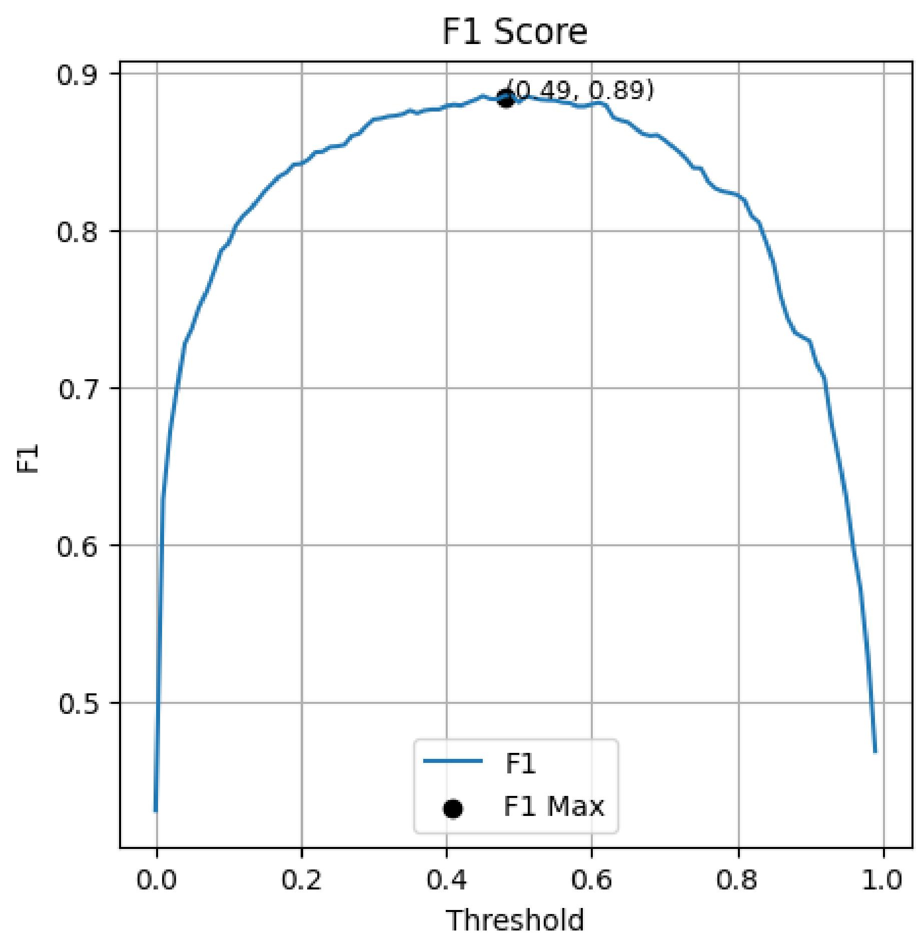
At which threshold F1 is maximal?

- 0.12
- 0.32
- **0.52**
- 0.72

In [ ]:
```
# Compute precision and recall using your function
thresholds = np.arange(0.0, 1.0, 0.01)
_, _, _, precision, recall = roc_prec_recall(   Y_pred_proba, Y_val,
                                                thresholds, feature_selection=False)

# Compute F1 scores
precision = np.array(precision)
recall = np.array(recall)
F1 = 2 * (precision * recall) / (precision + recall)
```

In [ ]:
```
# Compute precision and recall using your function
thresholds = np.arange(0.0, 1.0, 0.01)
_, _, _, precision, recall = roc_prec_recall(   Y_pred_proba, Y_val,
                                                thresholds, feature_selection=False)

# Compute F1 scores
precision = np.array(precision)
recall = np.array(recall)
F1 = 2 * (precision * recall) / (precision + recall)
```

```
# F1 maximum
idx = np.argmax(F1)
max_threshold = thresholds[idx]
max_f1 = F1[idx]

plt.figure(figsize=(5, 5))
plt.plot(thresholds, F1, label=f'F1')
plt.scatter(intersect_threshold,intersect_recall, color = 'black', label='F1 Max')
plt.text(   intersect_threshold, intersect_recall,
            f'({max_threshold:.2f}, {max_f1:.2f})', color = 'black', fontsize=9)

# Settings
plt.legend()
plt.xlabel('Threshold')
plt.ylabel('F1')
plt.title('F1 Score')
plt.grid(True)
plt.show()
```



4. k-Fold Cross-Validation

Cross-validation is a technique where we divide the dataset, $\mathbf{D}$, into k parts of equal size, referred to as folds (e.g., $\mathbf{D}_1, \mathbf{D}_2, \cdots, \mathbf{D}_k$). Each fold, $\mathbf{D}_i$, acts as a testing set once, while the remaining folds serve as the training set.

To illustrate, let's say we have a dataset and we choose $k = 3$. The dataset is split into 3 subsets or folds. In each iteration, one fold is used as the test set, and the remaining two as the training set, as shown below:

No description has been provided for this image

Once the dataset, $\mathbf{D}$, is shuffled, it's divided into k approximately equal segments or folds. In each validation round, a specific fold, $\mathbf{D}_i$, is selected as the testing set. The model is then trained using the collective data from the remaining folds and subsequently evaluated on the selected testing set, $\mathbf{D}_i$.

It's worth noting that in standard k-fold CV, the data is divided once and not reshuffled. However, in techniques like repeated k-fold CV, reshuffling and re-splitting into k-folds is done multiple times to enhance reliability.

- **Question 5: 5-Fold CV**

Use the `KFold` class from Scikit-Learn to evaluate our model on 5 different folds:

```
KFold(n_splits=5, shuffle=True, random_state=1)
```

- Iterate over different folds of `df_full_train`
- Split the data into train and validation
- Train the model on train with these parameters: `LogisticRegression(solver='liblinear', C=1.0, max_iter=1000)`
- Use AUC to evaluate the model on validation

How large is standard deviation of the scores across different folds?

- **0.003**

- 0.030
- 0.090
- 0.140

In [ ]:
```python
def train(df, y, drop_feature = None, C = 1.0, max_iter = 1000):


    if drop_feature is not None:
        df = df.drop(columns=[drop_feature])
    df_dict = df.to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    dv.fit(df_dict)

    x_train = dv.transform(df_dict)

    model = LogisticRegression(solver='liblinear', C = C, max_iter= max_iter)
    model.fit(x_train, y)

    return dv, model


def predict(df, dv, model):
    df_dict = df.to_dict(orient='records')

    X = dv.transform(df_dict)

    y_pred = model.predict_proba(X)[:, 1]

    return y_pred
```

In [ ]:
```python
kfold = KFold(n_splits = 5 , shuffle = True)

aucs = []
for train_idx, val_idx in kfold.split(df_full_train):

    # Split data in train and validation
    df_train = df_full_train.iloc[train_idx]
    Y_train = df_train['above_avg'].values

    df_val = df_full_train.iloc[val_idx]
    Y_val = df_val['above_avg'].values

    # train on train set and predict model on validation
    dv, model = train(df_train, Y_train, drop_feature = 'above_avg')
    Y_pred_proba = predict(df_val, dv, model)

    # compute the AUC score and store it in the list
    auc_value, _, _, _, _ = roc_prec_recall(Y_pred_proba, Y_val, feature_selection= False)
    aucs.append(auc_value)


# AUC for each fold
print('AUC folders:\n',np.array(aucs).round(3))

# Mean and standard deviation of AUCs across all folds
print('AUC stats = %0.3f ± %0.3f' % (np.mean(aucs), np.std(aucs)))
```

```
AUC folders:
 [0.976 0.977 0.978 0.977 0.985]
AUC stats = 0.979 ± 0.003
```

5. Tuning the parameter `C`

In logistic regression, a component known as the regularization term is incorporated into the cost function. This term operates by introducing a penalty to the cost function, analogous to the L1 regularization for linear regression. The penalty term helps prevent the model from overfitting by limiting the magnitude of the model parameters.

The strength of the regularization is controlled by the C parameter. A smaller C means stronger regularization, leading to smaller model parameters, whereas a larger C corresponds to weaker regularization, allowing the model parameters to take larger values.

Thus, by adjusting C, we can control the balance between keeping the model simple to avoid overfitting and allowing it to fit the training data as closely as possible.

- **Question 6: Hyperparemeter Tuning**

Now let's use 5-Fold cross-validation to find the best parameter `C`

- Iterate over the following `C` values: `[0.01, 0.1, 0.5, 10]`

- Initialize `KFold` with the same parameters as previously
- Use these parametes for the model: `LogisticRegression(solver='liblinear', C=C, max_iter=1000)`
- Compute the mean score as well as the std (round the mean and std to 3 decimal digits)

Which `C` leads to the best mean score?

- 0.01
- 0.1
- 0.5
- **10**

In [ ]:
```
kfold = KFold(n_splits = 5 , shuffle = True, random_state = 1)

for C in [0.01, 0.1, 0.5, 10]:
    aucs = []
    for train_idx, val_idx in kfold.split(df_full_train):

        # Split data in train and validation
        df_train = df_full_train.iloc[train_idx]
        Y_train = df_train['above_avg'].values

        df_val = df_full_train.iloc[val_idx]
        Y_val = df_val['above_avg'].values

        # train on train set and predict model on validation
        dv, model = train(df_train, Y_train, drop_feature = 'above_avg', C = C)
        Y_pred_proba = predict(df_val, dv, model)

        # compute the AUC score and store it in the list
        auc_value, _, _, _, _ = roc_prec_recall(    Y_pred_proba, Y_val,
                                                    feature_selection= False)
        aucs.append(auc_value)
    print('C=%s, auc = %0.5f ± %0.5f' % (C, np.mean(aucs), np.std(aucs)))
```

```
C=0.01, auc = 0.95162 ± 0.00253
C=0.1, auc = 0.97180 ± 0.00159
C=0.5, auc = 0.97694 ± 0.00162
C=10, auc = 0.98134 ± 0.00326
```