In [1]:

```
import os

import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import (accuracy_score,
                             f1_score,
                             roc_auc_score,
                             roc_curve,
                             confusion_matrix)
from sklearn.model_selection import (cross_val_score,
                                     GridSearchCV,
                                     RandomizedSearchCV,
                                     learning_curve,
                                     validation_curve,
                                     train_test_split)
from sklearn.pipeline import make_pipeline
from sklearn.utils import resample
from warnings import filterwarnings

%matplotlib inline
sns.set_context("notebook")
plt.style.use("fivethirtyeight")
filterwarnings("ignore")
```

In [2]:

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from sklearn.metrics import confusion_matrix, roc_auc_score, roc_curve
from sklearn.metrics import f1_score, roc_auc_score, precision_recall_curve, roc_curve


def plot_conf_matrix_and_roc(estimator, X, y, figure_size=(16, 6)):
    """
    Plot both confusion matrix and ROC curce on the same figure.
    Parameters:
    -----------
    estimator : sklearn.estimator
        model to use for predicting class probabilities.
    X : array_like
        data to predict class probabilities.
    y : array_like
        true label vector.
    figure_size : tuple (optional)
        size of the figure.
    Returns:
    --------
    plot : matplotlib.pyplot
        plot confusion matrix and ROC curve.
    """
    # Compute tpr, fpr, auc and confusion matrix
    fpr, tpr, thresholds = roc_curve(y, estimator.predict_proba(X)[:, 1])
    auc = roc_auc_score(y, estimator.predict_proba(X)[:, 1])
    conf_mat_rf = confusion_matrix(y, estimator.predict(X))

    # Define figure size and figure ratios
    plt.figure(figsize=figure_size)
    gs = GridSpec(1, 2, width_ratios=(1, 2))

    # Plot confusion matrix
    ax0 = plt.subplot(gs[0])
    ax0.matshow(conf_mat_rf, cmap=plt.cm.Reds, alpha=0.2)

    for i in range(2):
        for j in range(2):
            ax0.text(x=j, y=i, s=conf_mat_rf[i, j], ha="center", va="center")
    plt.title("Confusion matrix", y=1.1, fontdict={"fontsize": 20})
    plt.xlabel("Predicted", fontdict={"fontsize": 14})
    plt.ylabel("Actual", fontdict={"fontsize": 14})


    # Plot ROC curce
    ax1 = plt.subplot(gs[1])
    ax1.plot(fpr, tpr, label="auc = {:.3f}".format(auc))
    plt.title("ROC curve", y=1, fontdict={"fontsize": 20})
    ax1.plot([0, 1], [0, 1], "r--")
    plt.xlabel("False positive rate", fontdict={"fontsize": 16})
    plt.ylabel("True positive rate", fontdict={"fontsize": 16})
    plt.legend(loc="lower right", fontsize="medium");


def plot_roc(estimators, X, y, figure_size=(16, 6)):
    """
    Plot both confusion matrix and ROC curce on the same figure.
    Parameters:
```

```
    -----------
    estimators : dict
        key, value for model name and sklearn.estimator to use for predicting
        class probabilities.
    X : array_like
        data to predict class probabilities.
    y : array_like
        true label vector.
    figure_size : tuple (optional)
        size of the figure.
    Returns:
    --------
    plot : matplotlib.pyplot
        plot confusion matrix and ROC curve.
    """
    plt.figure(figsize=figure_size)
    for estimator in estimators.keys():
        # Compute tpr, fpr, auc and confusion matrix
        fpr, tpr, thresholds = roc_curve(y, estimators[estimator].predict_proba(X)[:, 1])
        auc = roc_auc_score(y, estimators[estimator].predict_proba(X)[:, 1])

        # Plot ROC curce
        plt.plot(fpr, tpr, label="{}: auc = {:.3f}".format(estimator, auc))
        plt.title("ROC curve", y=1, fontdict={"fontsize": 20})
        plt.legend(loc="lower right", fontsize="medium")

    plt.plot([0, 1], [0, 1], "--")
    plt.xlabel("False positive rate", fontdict={"fontsize": 16})
    plt.ylabel("True positive rate", fontdict={"fontsize": 16});


def plot_roc_and_pr_curves(models, X_train, y_train, X_valid, y_valid, roc_title, pr_title,
    """
    Plot roc and PR curves for all models.

    Arguments
    ---------
    models : list
        list of all models.
    X_train : list or 2d-array
        2d-array or list of training data.
    y_train : list
        1d-array or list of training labels.
    X_valid : list or 2d-array
        2d-array or list of validation data.
    y_valid : list
        1d-array or list of validation labels.
    roc_title : str
        title of ROC curve.
    pr_title : str
        title of PR curve.
    labels : list
        label of each model to be displayed on the legend.
    """
    fig, axes = plt.subplots(1, 2, figsize=(14, 6))

    if not isinstance(X_train, list):
        for i, model in enumerate(models):
            model_fit = model.fit(X_train, y_train)
            model_probs = model.predict_proba(X_valid)[:, 1:]
            model_preds = model.predict(X_valid)
```

```
            model_auc_score = roc_auc_score(y_valid, model_probs)
            # model_f1_score = f1_score(y_valid, model_preds)
            fpr, tpr, _ = roc_curve(y_valid, model_probs)
            precision, recall, _ = precision_recall_curve(y_valid, model_probs)
            axes[0].plot(fpr, tpr, label=f"{labels[i]}, auc = {model_auc_score:.3f}")
            axes[1].plot(recall, precision, label=f"{labels[i]}")
    else:
        for i, model in enumerate(models):
            model_fit = model.fit(X_train[i], y_train[i])
            model_probs = model.predict_proba(X_valid[i])[:, 1:]
            model_preds = model.predict(X_valid[i])
            model_auc_score = roc_auc_score(y_valid[i], model_probs)
            # model_f1_score = f1_score(y_valid[i], model_preds)
            fpr, tpr, _ = roc_curve(y_valid[i], model_probs)
            precision, recall, _ = precision_recall_curve(y_valid[i], model_probs)
            axes[0].plot(fpr, tpr, label=f"{labels[i]}, auc = {model_auc_score:.3f}")
            axes[1].plot(recall, precision, label=f"{labels[i]}")
    axes[0].legend(loc="lower right")
    axes[0].set_xlabel("FPR")
    axes[0].set_ylabel("TPR")
    axes[0].set_title(roc_title)
    axes[1].legend()
    axes[1].set_xlabel("recall")
    axes[1].set_ylabel("precision")
    axes[1].set_title(pr_title)
    plt.tight_layout()
```

In [3]:

```
# Load the data
df = pd.read_csv("C:/Users/hp/Desktop/hr_data.csv")

# Check both the datatypes and if there is missing values
print("\033[1m" + "\033[94m" + "Data types:\n" + 11 * "-")
print("\033[30m" + "{}\n".format(df.dtypes))
print("\033[1m" + "\033[94m" + "Sum of null values in each column:\n" + 35 * "-")
print("\033[30m" + "{}".format(df.isnull().sum()))
df.head()
```

```
Data types:
-----------
satisfaction_level       float64
last_evaluation          float64
number_project             int64
average_montly_hours       int64
time_spend_company         int64
Work_accident              int64
promotion_last_5years      int64
department                object
salary                    object
left                       int64
dtype: object

Sum of null values in each column:
-----------------------------------
satisfaction_level       0
last_evaluation          0
number_project           0
average_montly_hours     0
time_spend_company       0
Work_accident            0
promotion_last_5years    0
department               0
salary                   0
left                     0
dtype: int64
```

Out[3]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_compa
0	0.38	0.53	2	157	
1	0.80	0.86	5	262	
2	0.11	0.88	7	272	
3	0.72	0.87	5	223	
4	0.37	0.52	2	159	

In [4]:

```
# Rename sales feature into department
df = df.rename(columns={"sales": "department"})

# Map salary into integers
salary_map = {"low": 0, "medium": 1, "high": 2}
df["salary"] = df["salary"].map(salary_map)

# Create dummy variables for department feature
df = pd.get_dummies(df, columns=["department"], drop_first=True)
df.head()
```

Out[4]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_compa
0	0.38	0.53	2	157	
1	0.80	0.86	5	262	
2	0.11	0.88	7	272	
3	0.72	0.87	5	223	
4	0.37	0.52	2	159	

In [5]:

```
df.columns[df.columns != "left"].shape
```

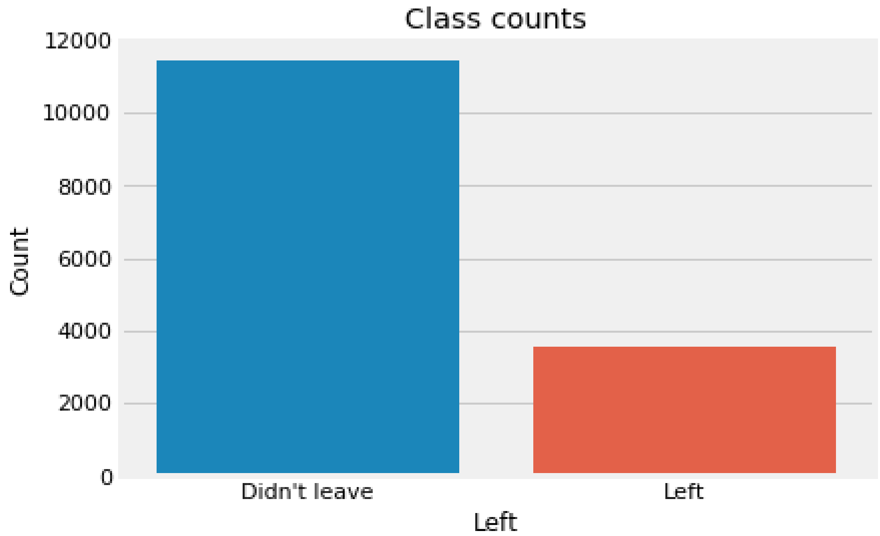
Out[5]:

```
(17,)
```

In [6]:

```
# Get number of positve and negative examples
pos = df[df["left"] == 1].shape[0]
neg = df[df["left"] == 0].shape[0]
print("Positive examples = {}".format(pos))
print("Negative examples = {}".format(neg))
print("Proportion of positive to negative examples = {:.2f}%".format((pos / neg) * 100))
sns.countplot(df["left"])
plt.xticks((0, 1), ["Didn't leave", "Left"])
plt.xlabel("Left")
plt.ylabel("Count")
plt.title("Class counts");
```

```
Positive examples = 3571
Negative examples = 11428
Proportion of positive to negative examples = 31.25%
```



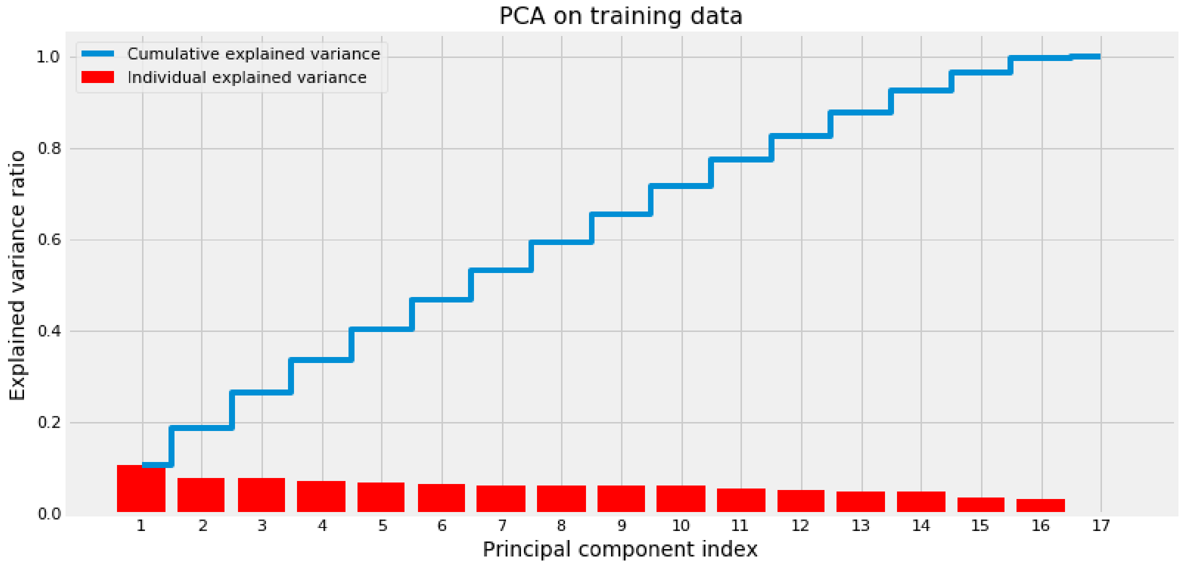
In [7]:

```
# Convert dataframe into numpy objects and split them into
# train and test sets: 80/20
X = df.loc[:, df.columns != "left"].values
y = df.loc[:, df.columns == "left"].values.flatten()

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=1)

# Upsample minority class
X_train_u, y_train_u = resample(X_train[y_train == 1],
                                y_train[y_train == 1],
                                replace=True,
                                n_samples=X_train[y_train == 0].shape[0],
                                random_state=1)
X_train_u = np.concatenate((X_train[y_train == 0], X_train_u))
y_train_u = np.concatenate((y_train[y_train == 0], y_train_u))

# Downsample majority class
X_train_d, y_train_d = resample(X_train[y_train == 0],
                                y_train[y_train == 0],
                                replace=True,
                                n_samples=X_train[y_train == 1].shape[0],
                                random_state=1)
X_train_d = np.concatenate((X_train[y_train == 1], X_train_d))
y_train_d = np.concatenate((y_train[y_train == 1], y_train_d))

print("Original shape:", X_train.shape, y_train.shape)
print("Upsampled shape:", X_train_u.shape, y_train_u.shape)
print("Downsampled shape:", X_train_d.shape, y_train_d.shape)
```

```
Original shape: (11999, 17) (11999,)
Upsampled shape: (18284, 17) (18284,)
Downsampled shape: (5714, 17) (5714,)
```

In [8]:

```
# Build PCA using standarized trained data
pca = PCA(n_components=None, svd_solver="full")
pca.fit(StandardScaler().fit_transform(X_train))
cum_var_exp = np.cumsum(pca.explained_variance_ratio_)
plt.figure(figsize=(12, 6))
plt.bar(range(1, 18), pca.explained_variance_ratio_, align="center",
        color='red', label="Individual explained variance")
plt.step(range(1, 18), cum_var_exp, where="mid", label="Cumulative explained variance")
plt.xticks(range(1, 18))
plt.legend(loc="best")
plt.xlabel("Principal component index", {"fontsize": 14})
plt.ylabel("Explained variance ratio", {"fontsize": 14})
plt.title("PCA on training data", {"fontsize": 16});
```



In [9]:

```
cum_var_exp
```

Out[9]:

```
array([0.1078147 , 0.18756726, 0.26523205, 0.33604446, 0.4036422 ,
       0.46807506, 0.53094596, 0.59334034, 0.65535106, 0.71691288,
       0.77413324, 0.82651546, 0.87672244, 0.92515346, 0.96216602,
       0.99429813, 1.        ])
```

In [16]:

```
# Build random forest classifier
methods_data = {"Original": (X_train, y_train),
                "Upsampled": (X_train_u, y_train_u),
                "Downsampled": (X_train_d, y_train_d)}

for method in methods_data.keys():
    pip_rf = make_pipeline(StandardScaler(),
                           RandomForestClassifier(n_estimators=500,
                                                  class_weight="balanced",
                                                  random_state=123))

    hyperparam_grid = {
        "randomforestclassifier__n_estimators": [10, 50, 100, 500],
        "randomforestclassifier__max_features": ["sqrt", "log2", 0.4, 0.5],
        "randomforestclassifier__min_samples_leaf": [1, 3, 5],
        "randomforestclassifier__criterion": ["gini", "entropy"]}

    gs_rf = GridSearchCV(pip_rf,
                         hyperparam_grid,
                         scoring="f1",
                         cv=10,
                         n_jobs=-1)

    gs_rf.fit(methods_data[method][0], methods_data[method][1])

    print("\033[1m" + "\033[0m" + "The best hyperparameters for {} data:".format(method))
    for hyperparam in gs_rf.best_params_.keys():
        print(hyperparam[hyperparam.find("__") + 2:], ": ", gs_rf.best_params_[hyperparam])

    print("\033[1m" + "\033[94m" + "Best 10-folds CV f1-score: {:.2f}%.".format((gs_rf.best
```

```
The best hyperparameters for Original data:
criterion :  gini
max_features :  0.5
min_samples_leaf :  1
n_estimators :  500
Best 10-folds CV f1-score: 98.19%.
The best hyperparameters for Upsampled data:
criterion :  entropy
max_features :  0.4
min_samples_leaf :  1
n_estimators :  50
Best 10-folds CV f1-score: 99.80%.
The best hyperparameters for Downsampled data:
criterion :  entropy
max_features :  0.4
min_samples_leaf :  1
n_estimators :  500
Best 10-folds CV f1-score: 98.42%.
```

In [10]:

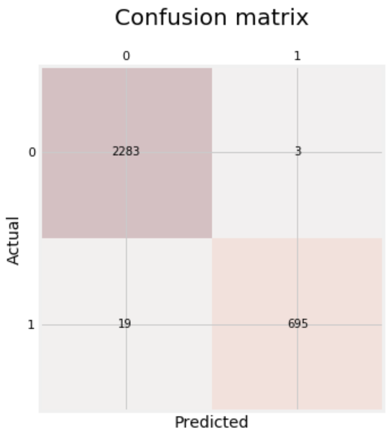
```
X_train_u[y_train_u == 0].shape, X_train_u[y_train_u == 1].shape
```

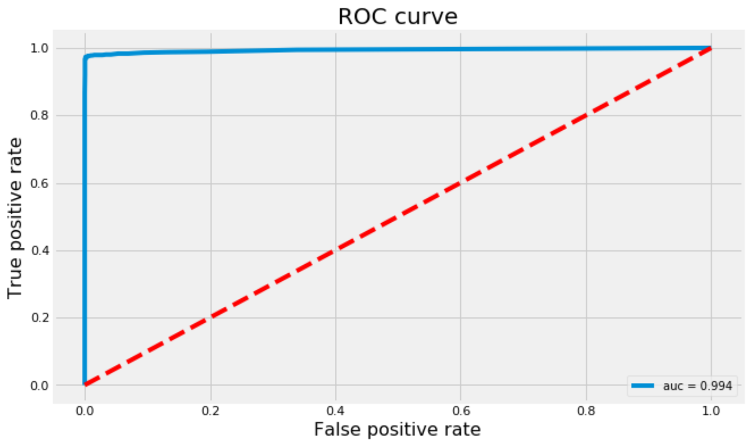
Out[10]:

```
((9142, 17), (9142, 17))
```

In [11]:

```
# Reassign original training data to upsampled data
X_train, y_train = np.copy(X_train_u), np.copy(y_train_u)

# Delete original and downsampled data
del X_train_u, y_train_u, X_train_d, y_train_d

# Refit RF classifier using best params
clf_rf = make_pipeline(StandardScaler(),
                       RandomForestClassifier(n_estimators=50,
                                              criterion="entropy",
                                              max_features=0.4,
                                              min_samples_leaf=1,
                                              class_weight="balanced",
                                              n_jobs=-1,
                                              random_state=123))



clf_rf.fit(X_train, y_train)

# Plot confusion matrix and ROC curve
plot_conf_matrix_and_roc(clf_rf, X_test, y_test)
```





In [ ]:

```
# Build Gradient Boosting classifier
pip_gb = make_pipeline(StandardScaler(),
                       GradientBoostingClassifier(loss="deviance",
                                                  random_state=123))

hyperparam_grid = {"gradientboostingclassifier__max_features": ["log2", 0.5],
                   "gradientboostingclassifier__n_estimators": [100, 300, 500],
                   "gradientboostingclassifier__learning_rate": [0.001, 0.01, 0.1],
                   "gradientboostingclassifier__max_depth": [1, 2, 3]}

gs_gb = GridSearchCV(pip_gb,
                     param_grid=hyperparam_grid,
                     scoring="f1",
                     cv=10,
                     n_jobs=-1)

gs_gb.fit(X_train, y_train)

print("\033[1m" + "\033[0m" + "The best hyperparameters:")
print("-" * 25)
for hyperparam in gs_gb.best_params_.keys():
    print(hyperparam[hyperparam.find("__") + 2:], ": ", gs_gb.best_params_[hyperparam])

print("\033[1m" + "\033[94m" + "Best 10-folds CV f1-score: {:.2f}%.".format((gs_gb.best_scc
```

In [ ]:

```
# Plot confusion matrix and ROC curve
plot_conf_matrix_and_roc(gs_gb, X_test, y_test)
```

In [ ]:

```
# Build KNN classifier
pip_knn = make_pipeline(StandardScaler(), KNeighborsClassifier())
hyperparam_range = range(1, 20)

gs_knn = GridSearchCV(pip_knn,
                      param_grid={"kneighborsclassifier__n_neighbors": hyperparam_range,
                                  "kneighborsclassifier__weights": ["uniform", "distance"]}
                      scoring="f1",
                      cv=10,
                      n_jobs=-1)

gs_knn.fit(X_train, y_train)


print("\033[1m" + "\033[0m" + "The best hyperparameters:")
print("-" * 25)
for hyperparam in gs_knn.best_params_.keys():
    print(hyperparam[hyperparam.find("__") + 2:], ": ", gs_knn.best_params_[hyperparam])

print("\033[1m" + "\033[94m" + "Best 10-folds CV f1-score: {:.2f}%.".format((gs_knn.best_sc
```

In [ ]:

```
plot_conf_matrix_and_roc(gs_knn, X_test, y_test)
```

In [ ]:

```
# Build logistic model classifier
pip_logmod = make_pipeline(StandardScaler(),
                           LogisticRegression(class_weight="balanced"))

hyperparam_range = np.arange(0.5, 20.1, 0.5)

hyperparam_grid = {"logisticregression__penalty": ["l1", "l2"],
                   "logisticregression__C":  hyperparam_range,
                   "logisticregression__fit_intercept": [True, False]
                  }

gs_logmodel = GridSearchCV(pip_logmod,
                           hyperparam_grid,
                           scoring="accuracy",
                           cv=2,
                           n_jobs=-1)

gs_logmodel.fit(X_train, y_train)

print("\033[1m" + "\033[0m" + "The best hyperparameters:")
print("-" * 25)
for hyperparam in gs_logmodel.best_params_.keys():
    print(hyperparam[hyperparam.find("__") + 2:], ": ", gs_logmodel.best_params_[hyperparam

print("\033[1m" + "\033[94m" + "Best 10-folds CV f1-score: {:.2f}%.".format((gs_logmodel.be
```

In [ ]:

```
plot_conf_matrix_and_roc(gs_logmodel, X_test, y_test)
```

In [ ]:

```
# Build SVM classifier
clf_svc = make_pipeline(StandardScaler(),
                        SVC(C=0.01,
                            gamma=0.1,
                            kernel="poly",
                            degree=5,
                            coef0=10,
                            probability=True))

clf_svc.fit(X_train, y_train)

svc_cv_scores = cross_val_score(clf_svc,
                                X=X_train,
                                y=y_train,
                                scoring="f1",
                                cv=10,
                                n_jobs=-1)

# Print CV
print("\033[1m" + "\033[94m" + "The 10-folds CV f1-score is: {:.2f}%".format(
       np.mean(svc_cv_scores) * 100))
```

In [ ]:

```
plot_conf_matrix_and_roc(clf_svc, X_test, y_test)
```

In [ ]:

```
# Plot ROC curves for all classifiers
estimators = {"RF": clf_rf,
              "LR": gs_logmodel,
              "SVC": clf_svc,
              "GBT": gs_gb,
              "KNN": gs_knn}
plot_roc(estimators, X_test, y_test, (12, 8))

# Print out accuracy score on test data
print("The accuracy rate and f1-score on test data are:")
for estimator in estimators.keys():
    print("{}: {:.2f}%, {:.2f}%.".format(estimator,
        accuracy_score(y_test, estimators[estimator].predict(X_test)) * 100,
         f1_score(y_test, estimators[estimator].predict(X_test)) * 100))
```

In [ ]:

```
# Refit RF classifier
clf_rf = RandomForestClassifier(n_estimators=50,
                                criterion="entropy",
                                max_features=0.4,
                                min_samples_leaf=1,
                                class_weight="balanced",
                                n_jobs=-1,
                                random_state=123)


clf_rf.fit(StandardScaler().fit_transform(X_train), y_train)

# Plot features importance
importances = clf_rf.feature_importances_
indices = np.argsort(clf_rf.feature_importances_)[::-1]
plt.figure(figsize=(12, 6))
plt.bar(range(1, 18), importances[indices], align="center")
plt.xticks(range(1, 18), df.columns[df.columns != "left"][indices], rotation=90)
plt.title("Feature Importance", {"fontsize": 16});
```