Отчёт о выполнении домашней контрольной работы №2, вариант №1.  
Выполнил студент группы M9120-09.04.01иибд  
Глушков Владимир Константинович.

Расчёты и построение графиков для выполнения данной контрольной работы выполнялись на языке программирования Python с использование следующих библиотек: matplotlib (построение графиков), numpy, statsmodels

Для описания выполнения заданий используются выдержки из кода, которые могут требовать импорта библиотек и функций, описанных в других выдержках из кода. Полный исходный код решения контрольной работы представлен в приложении.

# Задание № 1. Временные ряды

1. **Постройте график временного ряда**

 Выполним построение с помощью библиотеки matplotlib

|  |
| --- |
| def subtask1():  print(**"Start subtask #1"**)  data = getData()  dates = getDates(len(data))  plt.figure(figsize=(10, 4))  plt.xticks(rotation=90)  plt.plot(list(map(lambda x: x.strftime(**"%m.%y"**), dates)), data)  plt.ylabel(**"Объем экспорта тыс. долл"**)  plt.title(**"график временного ряда"**)  plt.show()  print(**"End subtask #1"**) |

1. **Постройте автокорреляционную функцию ACF (таблица, график)**

Вычислим функцию ACF по следующей формуле:

где — лаг (сдвиг во времени), который будем перебирать от 1 до 16.

def getMyCoefACF(x: np.ndarray, tau: int):  
 size = len(x)  
 a = np.array(x[tau:])  
 b = np.array(x[:size - tau])  
 return ((a \* b).mean() - a.mean() \* b.mean()) / (np.std(a) \* np.std(b))  
def ACF(x: np.ndarray, tau\_start, tau\_end):  
 coefs = []  
 for tau in range(tau\_start, tau\_end + 1):  
 coefs.append(getMyCoefACF(x, tau))  
 return coefs

Также проведём оценку значимости коэффициентов корреляции с помощью доверительных интервалов:

1. — где — функция квантиля стандартного квадратного распределения и — уровень значимости, который возьмём равным (зелёный график)

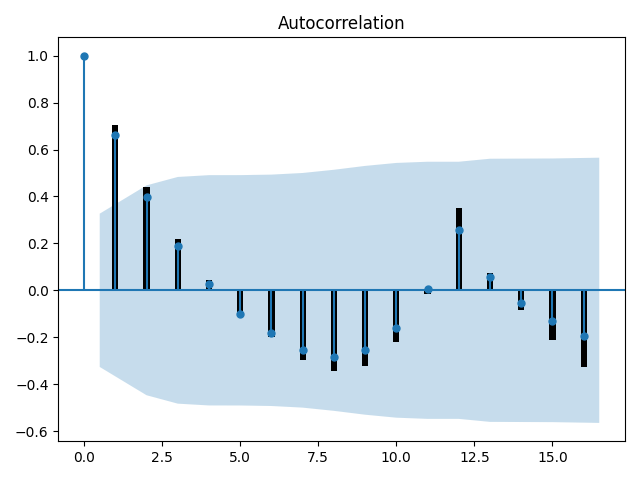
def confidence\_interval1(N, acf\_size):  
 return [[1.96 / np.sqrt(N)] \* acf\_size, [-1.96 / np.sqrt(N)] \* acf\_size]

1. . — где — отставание, равное лагу (синий график).

def confidence\_interval2(coefs, tau\_start, tau\_end, N):  
 confidence\_interval = []  
 for k in range(tau\_start, tau\_end + 1):  
 sum = 0  
 for i in range(k - 1):  
 sum += coefs[i] \* coefs[i]  
 confidence\_interval.append(1.96 \* np.sqrt((1 + 2 \* sum) / N))  
 return [confidence\_interval, [e \* -1 for e in confidence\_interval]]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Лаг | ACF | Лаг | ACF |
| 1 | 0.70 | 9 | -0.32 |
| 2 | 0.44 | 10 | -0.22 |
| 3 | 0.22 | 11 | -0.01 |
| 4 | 0.04 | 12 | 0.35 |
| 5 | -0.10 | 13 | 0.07 |
| 6 | -0.20 | 14 | -0.08 |
| 7 | -0.30 | 15 | -0.21 |
| 8 | -0.34 | 16 | -0.33 |

Дополнительно построим коррелограмму средствами библиотеки statsmodels (функция *plot\_acf*) и сравним полученные коэффициенты.

На основе построенных графиков и доверительных интервалов можно сделать вывод, что коэффициенты при лагах 1 не является значимым. Наибольший коэффициент автокорреляции получается при лаге 2, значит ряд имеет помимо тенденции периодические колебания с периодом 2

def subtask2():  
 print(**"Start subtask #2"**)  
 data = getData()  
 tau\_start = 1  
 tau\_end = 16  
 lags = list(range(tau\_start, tau\_end + 1))  
 acf\_res = ACF(data, tau\_start, tau\_end)  
 plt.bar(lags, acf\_res, width=0.2, color=**"black"**)  
 plt.xticks(lags, lags)  
 N = len(data)  
 conf1 = confidence\_interval1(N, len(acf\_res))  
 conf2 = confidence\_interval2(acf\_res, tau\_start, tau\_end, N)  
 plt.plot(lags, conf1[0], linewidth=1, color=**"green"**)  
 plt.plot(lags, conf1[1], linewidth=1, color=**"green"**)  
 plt.plot(lags, conf2[0], linewidth=1, color=**"blue"**)  
 plt.plot(lags, conf2[1], linewidth=1, color=**"blue"**)  
 plt.plot([1, 16], [0.0, 0.0], linewidth=1, color=**"black"**)  
 plt.ylabel(**"ACF"**)  
 plt.xlabel(**"Лаг"**)  
 plt.title(**"Коррелограмма"**)  
 plt.show()  
 print(**"Lags: "** + **" "**.join(list(map(str, range(tau\_start, tau\_end + 1)))))  
 print(**"ACF: "** + **" "**.join([**"{:.2f}"**.format(c) for c in acf\_res]))  
  
 plot\_acf(data)  
 plt.bar(lags, acf\_res, width=0.2, color=**"black"**)  
 plt.show()  
 print(**"End subtask #2"**)

1. **Стационарен ли временной ряд? Обоснуйте свой ответ используя графики и тесты.**

Построенный ранее график ACF характерен для нестационарных рядов. Убедимся в этом с помощью теста Дикки-Фуллера из библиотеки statsmodels.

def subtask3():  
 print(**"Start subtask #3"**)  
 data = getData()  
 m = len(data) // 2  
 test = adfuller(data)  
 print(**"Проведём тест Дики-Фуллера"**)  
 print(**"p-value: "**, test[1])  
 print(**"Critical values: "**, test[4])  
 if test[0] > test[4][**"5%"**]:  
 print(**"Eсть единичные корни, ряд не стационарен"**)  
 else:  
 print(**"Единичных корней нет, ряд стационарен"**)  
 print(**"End subtask #3"**)

Получаем:

adf: -2.20  
p-value: 0.21  
Critical values:   
 1%: -3.63  
 5%: -2.95  
 10%: -2.61

Полученная статистика больше, чем все критические значения, и  
поэтому можно утверждать, что ряд нестационарен.

1. **В зависимости от вида ACF постройте аддититивную или мультипликативную модель ряда.**

По виду графика временного ряда и ACF можно сказать, что подходит аддитивная модель.

Однако данные содержат малое количество колебаний, поэтому построим обе модели и выберем ту, где ошибка будет меньше.

def my\_decompose(data, df, period):  
 result = seasonal\_decompose(df, model=**'additive'**, period=period, extrapolate\_trend=**'freq'**)  
 trend = result.trend  
 season = result.seasonal  
  
 mae = ((data - (trend + season)) \*\* 2).mean()  
 return result, mae

print(**"Start subtask #4"**)  
data = getData()  
dates = getDates(len(data))  
  
df = pd.DataFrame({**"data"**: data}, index=pd.DatetimeIndex(dates))  
for period in range(2, 16):  
 decompose, mae = my\_decompose(data, df, period)  
 trend = decompose.trend  
 season = decompose.seasonal  
 print(**"period: "**, period)  
 print(**"**\t**% данных, объяс. моделью: "**, 1 - mae / data.var())  
 print(**"**\t**Абсолютная средняя ошибка: "**, np.abs(data - (trend)).mean())

Выполним декомпозицию с разными параметрами периода для аддитивной модели.

|  |  |  |
| --- | --- | --- |
| Период | % данных, объяснённых моделью | Абсолютная средняя ошибка |
| 2 | 0.93 | 8271.59 |
| 3 | 0.88 | 11403.88 |
| 4 | 0.83 | 16957.05 |
| 5 | 0.75 | 19821.46 |
| 6 | 0.76 | 25565.58 |
| 7 | 0.55 | 29829.64 |
| 8 | 0.28 | 36196.44 |
| 9 | 0.19 | 38183.22 |
| 10 | 0.11 | 42051.34 |
| 11 | 0.42 | 41579.25 |
| 12 | 0.83 | 40100.12 |
| 13 | 0.65 | 39995.67 |
| 14 | 0.61 | 41786.32 |
| 15 | 0.52 | 44349.70 |
| 16 | 0.35 | 48675.09 |

Выполним декомпозицию с разными параметрами периода для мультипликативной модели.

|  |  |  |
| --- | --- | --- |
| Период | % данных, объяснённых моделью | Абсолютная средняя ошибка |
| 2 | 0.92 | 8271.59 |
| 3 | 0.85 | 11403.88 |
| 4 | 0.77 | 16957.05 |
| 5 | 0.73 | 19821.46 |
| 6 | 0.63 | 25565.58 |
| 7 | 0.50 | 29829.64 |
| 8 | 0.16 | 36196.44 |
| 9 | 0.11 | 38183.22 |
| 10 | -0.01 | 42051.34 |
| 11 | 0.07 | 41579.25 |
| 12 | 0.18 | 40100.12 |
| 13 | 0.20 | 39995.67 |
| 14 | 0.22 | 41786.32 |
| 15 | 0.19 | 44349.70 |
| 16 | 0.12 | 48675.09 |

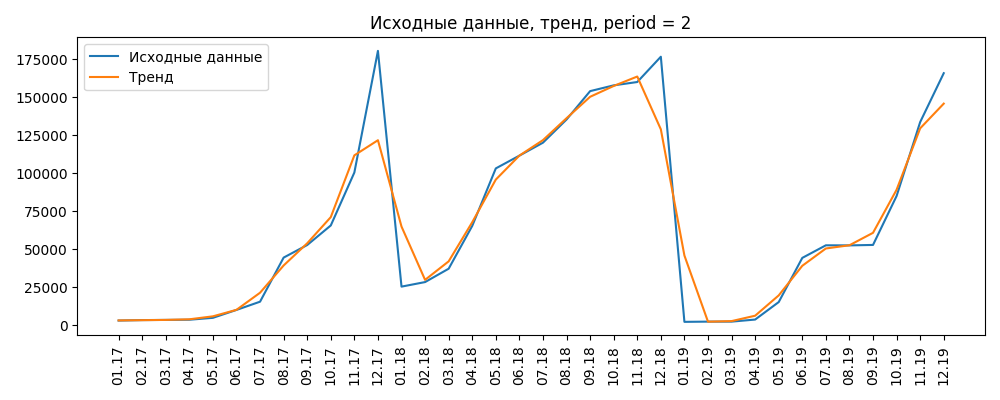
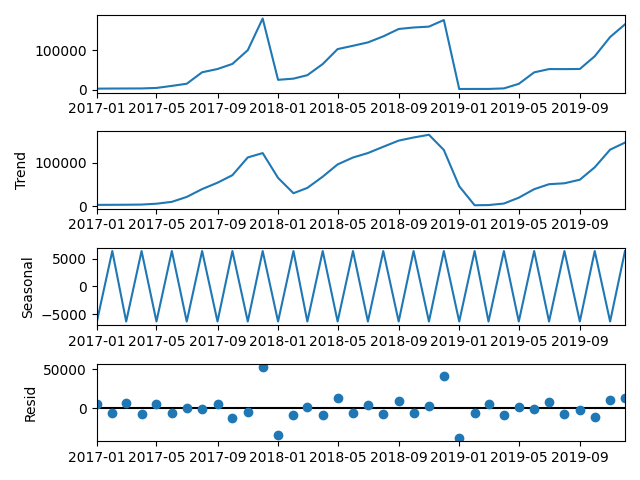
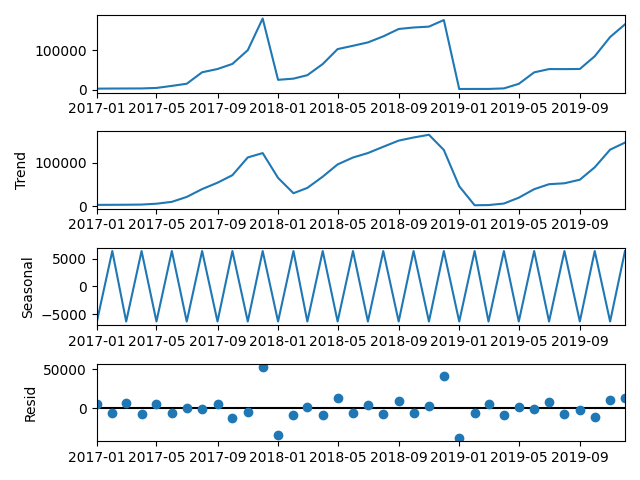
Следовательно аддитивная модель лучше описывает исходные данные с периодичностью 2, которая была определена по графику ACF ранее.

График сопоставления исходных данных и суммы T и S компонент ряда.

Декомпозиция модели с периодом 2.

1. **Обоснуйте выбор вида тренда (использовать графический анализ, среднюю ошибку аппроксимации, линейный коэффициент автокорреляции отклонений, статистическую значимость через F-критерий, t-критерий).**

Проанализируем тренд полученный на предыдущем шаге. Визуально тренд хорошо отражает исходные данные

Характеристики тренда:

1. Средняя ошибка апроксимации

print(**"Средняя абсолютная ошибка тенденции T: "**,  
 np.sqrt((np.square((data - decompose.trend) / len(data))).sum()) / data.mean())

Средняя абсолютная ошибка тенденции T: 0.04231590029911328

1. Линейный коэффициент автокорреляции

print(**"Линейный коэффициент корреляции: "**,  
 (((data - data.mean()) / data.std()) \* ((T - T.mean()) / T.std())).sum() / (len(data) - 1))

Линейный коэффициент корреляции: 0.9765220017371783

1. статистическую значимость через F-тест (критерий Фишера)

alpha = 0.05  
F = T.var() / data.var()  
df1 = len(data) - 1  
df2 = len(T) - 1  
p\_value = stats.f.cdf(F, df1, df2)  
print(**f"Вероятность превышения значения статистики:** {p\_value}**"**)  
if p\_value > alpha:  
 print(**'Нет достаточных оснований для отклонения нулевой гипотезы'**)  
else:  
 print(**'Есть достаточные основания для отклонения нулевой гипотезы'**)

Вероятность превышения значения статистики: 0.25514801833004913

Нет достаточных оснований для отклонения нулевой гипотезы

1. t-критерий

\_, p\_value = stats.ttest\_ind(T.dropna(), data, equal\_var=True)  
print(**"t критерий Стьюдента"**)  
print(**f"**\t**Вероятность превышения значения статистики:** {p\_value}**"**)  
if alpha < p\_value:  
 print(**"**\t**Нет достаточных оснований для отклонения нулевой гипотезы"**)  
else:  
 print(**"**\t**Есть достаточные основания для отклонения нулевой гипотезы"**)  
print(**"End subtask #5"**)

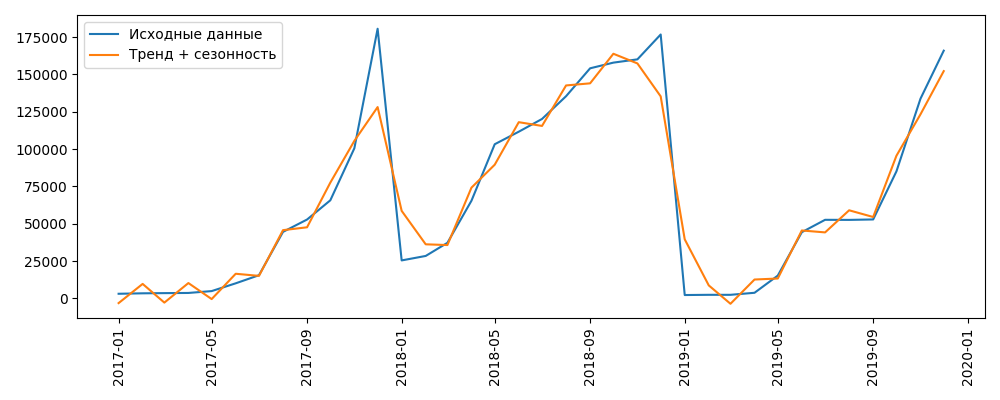
Вероятность превышения значения статистики: 0.9802427528626914  
Нет достаточных оснований для отклонения нулевой гипотезы

По всем параметрам полученный тренд хорошо описывает наши данные.

1. **Оцените качество модели через показатели средней абсолютной ошибки и среднего относительного отклонения.**

Построим среднюю абсолютную ошибку по формуле

def subtask6():  
 print(**"Start subtask #6"**)  
 data = getData()  
 dates = getDates(len(data))  
 df = pd.DataFrame({**"data"**: data}, index=pd.DatetimeIndex(dates))  
  
 decompose = seasonal\_decompose(df, model=**'additive'**, period=2, extrapolate\_trend=**'freq'**)  
  
 trend = decompose.trend  
 season = decompose.seasonal  
  
 mae = ((data - (trend + season)) \*\* 2).mean()  
 print(**f"MAE =** {mae}**"**)  
 print(**"MAE составляет "**, **"{:.2f}"**.format((mae / data.var()) \* 100), **"% дисперсии"**)  
 print(**"Модель объясняет "**, **"{:.2f}"**.format((1 - mae / data.var()) \* 100), **"% данных"**)  
  
 plt.figure(figsize=(10, 4))  
 plt.xticks(rotation=90)  
 plt.plot(df.index, df.values, label=**"Исходные данные"**)  
 plt.plot(df.index, trend + season, label=**"Тренд + сезонность"**)  
 plt.legend()  
 plt.show()  
  
 print(**"End subtask #6"**)

Средняя абсолютная ошибка составляет 6.82% дисперсии, следовательно созданная модель объясняет 93.18%.

1. **Выполните прогноз на 3 периода вперёд, оцените ошибку прогноза, постройте доверительный интервал прогноза для уровня значимости .**

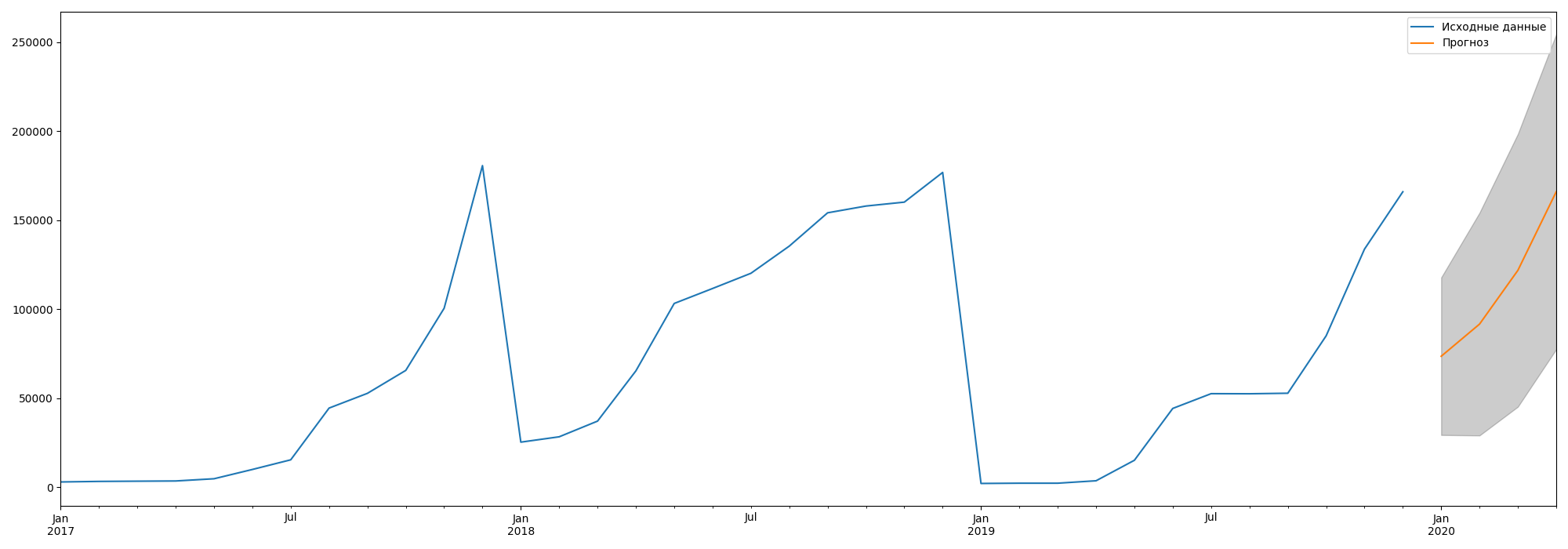
Построим прогноз по разложенному временному ряду. Для этого сделаем прогноз сезонной составляющей и сезонно скорректированной составляющей .

Сезонную компоненту спрогнозируем с помощью наивного метода и использованием значений за более ранний период. Сезонно скорректированные значения спрогнозируем с использованием тренда и значений остатков, т.е. также наивный метод.

data = getData()  
dates = pd.date\_range(**'2017-01-01'**, periods=36, freq=**'MS'**)  
df = pd.DataFrame({**"data"**: data}, index=pd.DatetimeIndex(dates))  
  
decompose = seasonal\_decompose(df,  
 model=**'additive'**,  
 extrapolate\_trend=**'freq'**)  
  
df\_reconstructed = pd.concat(  
 [decompose.seasonal,  
 decompose.trend,  
 decompose.resid,  
 decompose.trend + decompose.resid,  
 decompose.observed], axis=1)  
  
df\_reconstructed.columns = [**'seasonal'**, **'trend'**, **'remainders'**, **'seasonal\_adj'**, **'actual\_values'**]  
df\_reconstructed.dropna(inplace=True)  
  
df\_forecast = df\_reconstructed.iloc[-4:, :]  
df\_forecast = df\_forecast.set\_index(df\_forecast.index.shift(4))  
df\_forecast = df\_forecast.drop(**'actual\_values'**, axis=1)  
df\_forecast[[**'trend'**, **'remainders'**, **'seasonal\_adj'**]] = np.nan  
  
df\_forecast[**'trend'**] = df\_reconstructed.loc[df\_reconstructed.index[-1], **'trend'**]  
df\_forecast[**'remainders'**] = df\_reconstructed.loc[df\_reconstructed.index[-1], **'remainders'**]  
df\_forecast[**'seasonal\_adj'**] = df\_forecast[**'trend'**] + df\_forecast[**'remainders'**]  
df\_forecast[**'forecast'**] = df\_forecast[**'seasonal\_adj'**] + df\_forecast[**'seasonal'**]  
pd.set\_option(**'display.max\_columns'**, None)  
print(df\_forecast.head(n=3))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **seasonal** | **trend** | **remainders** | **seasonal\_adj** | **forecast** |
| 2020-01-01 | 17728.11 | 18944.67 | 36933.48 | 55878.15 | 73606.27 |
| 2020-02-01 | 35791.62 | 18944.67 | 36933.48 | 55878.15 | 91669.78 |
| 2020-03-01 | 65975.03 | 18944.67 | 36933.48 | 55878.15 | 121853.19 |

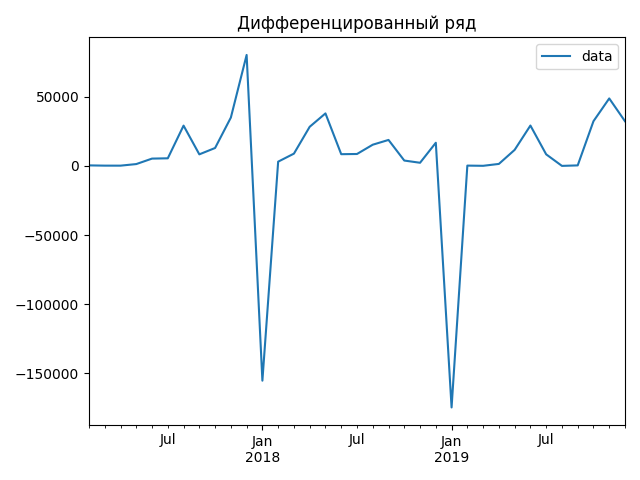
plt.rcParams.update({**'figure.figsize'**: (20, 7)})  
df\_reconstructed[**'actual\_values'**].plot()  
df\_forecast[**'forecast'**].plot()  
plt.show()

Вычислим доверительный интервал прогноза, в котором ожидается, что будет находиться с указанной вероятностью. Предположим, что ошибки прогноза распределены нормально. Тогда 95% интервал прогнозирования равен . df\_forecast[**'h'**] = range(1, 5)  
df\_forecast[**'std\_h'**] = resid\_std \* np.sqrt(df\_forecast[**'h'**])  
  
df\_forecast[**'lower\_adj'**] = (df\_forecast[**'seasonal\_adj'**] - 1.96 \* df\_forecast[**'std\_h'**])  
df\_forecast[**'upper\_adj'**] = (df\_forecast[**'seasonal\_adj'**] + 1.96 \* df\_forecast[**'std\_h'**])  
df\_forecast[**'lower'**] = df\_forecast[**'lower\_adj'**] + df\_forecast[**'seasonal'**]  
df\_forecast[**'upper'**] = df\_forecast[**'upper\_adj'**] + df\_forecast[**'seasonal'**]  
df\_forecast[[**'h'**, **'std\_h'**, **'lower'**, **'upper'**]]  
print(df\_forecast[[**'h'**, **'std\_h'**, **'lower'**, **'upper'**]].head(n=3))  
df\_reconstructed[**'actual\_values'**].plot(label=**"Исходные данные"**)  
df\_forecast[**'forecast'**].plot(label=**"Прогноз"**)  
plt.fill\_between(df\_forecast.index, df\_forecast[**'lower'**], df\_forecast[**'upper'**], color=**'k'**, alpha=.2)  
plt.legend()  
**plt.show()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | h | std\_h | lower | upper |
| 2020-01-01 | 1 | 22561.77 | 29385.19 | 117827.35 |
| 2020-02-01 | 2 | 31907.16 | 29131.73 | 154207.83 |
| 2020-03-01 | 3 | 39078.14 | 45260.03 | 198446.34 |

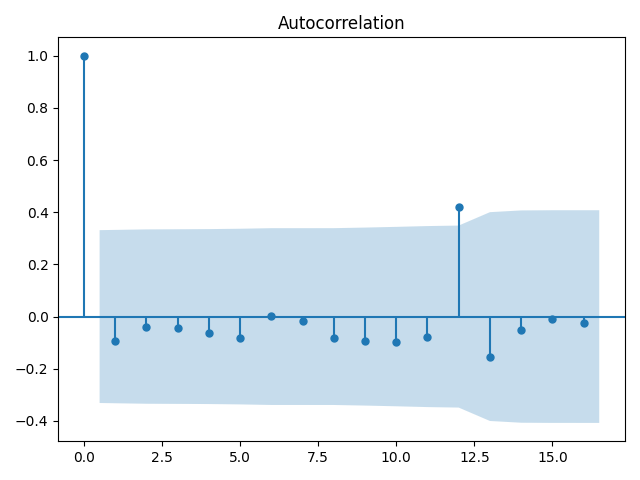
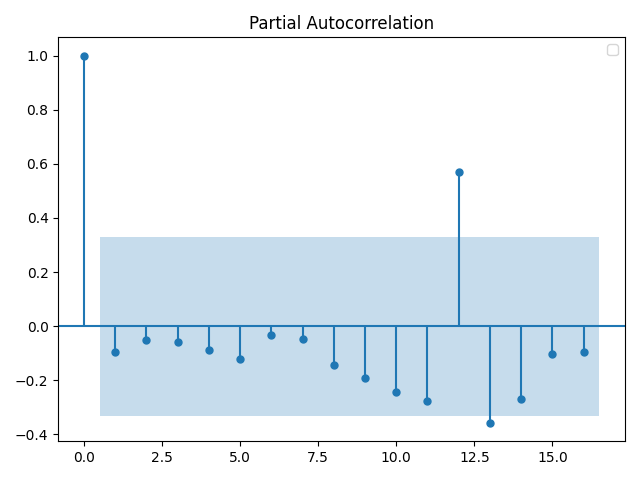
1. **Построить модель ARIMA(p,q,0), предварительно убедившись в степен и интеграции данного временного ряда и определив порядок авторегрессии**

Ранее с помощью теста Дикки-Фуллера из библиотеки statsmodels было определено, что ряд нестационарный, поэтому проведём дифференциацию ряда.

def diff(df):  
 diff\_ = df - df.shift(1)  
 diff\_.dropna(inplace=True)  
 return diff\_

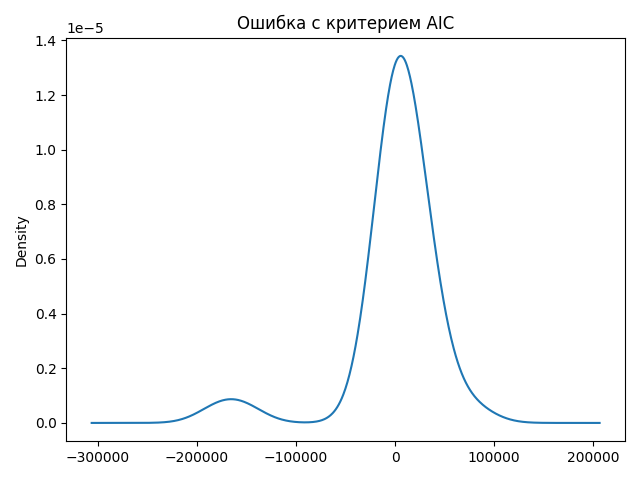
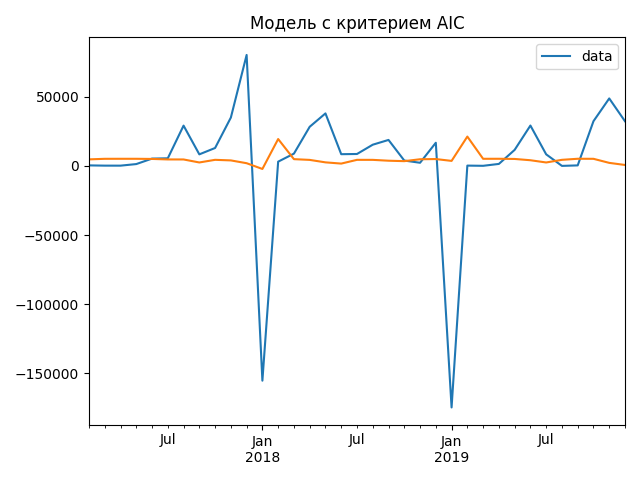
Проведём тест Дикки-Фуллера на дифференцированном ряду:

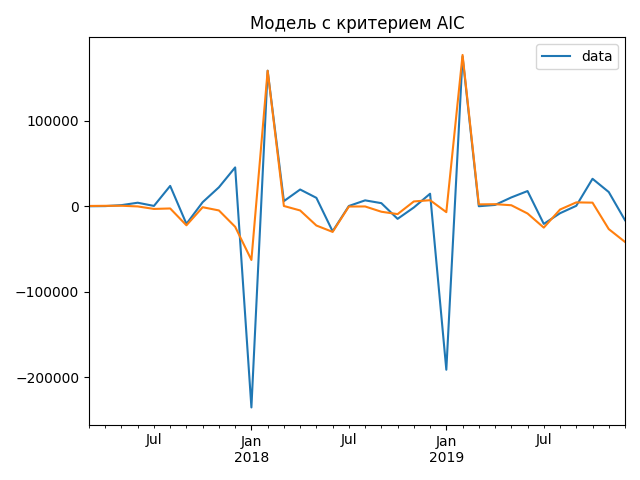
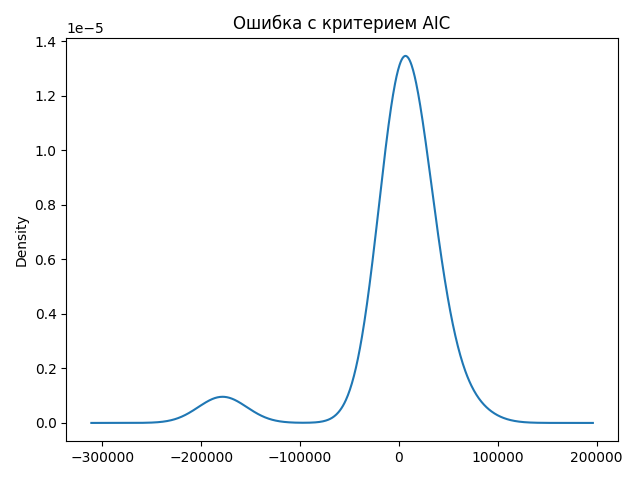
adf: -6.19  
p-value: 0.00  
Critical values:  
 1%: -3.64  
 5%: -2.95  
 10%: -2.61  
Единичных корней нет, ряд стационарен

Одной дифференциации было достаточно, ряд приведён к стационарному. Для определения параметров ARIMA построим ACF и PACF.

Подбор параметров выполним с помощью полного перебора минимизируя критерий Акаике AIC.

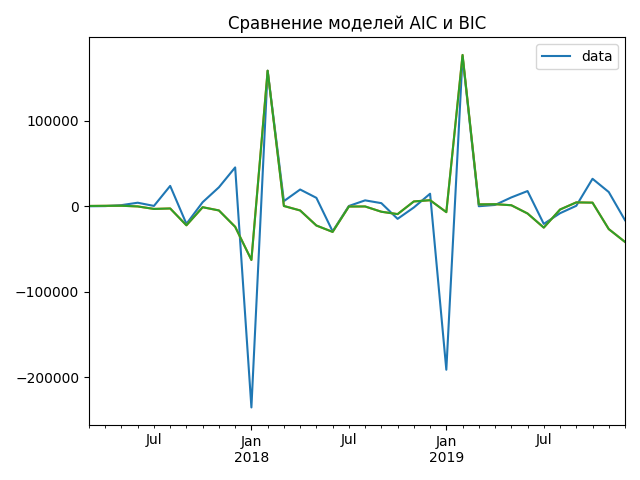
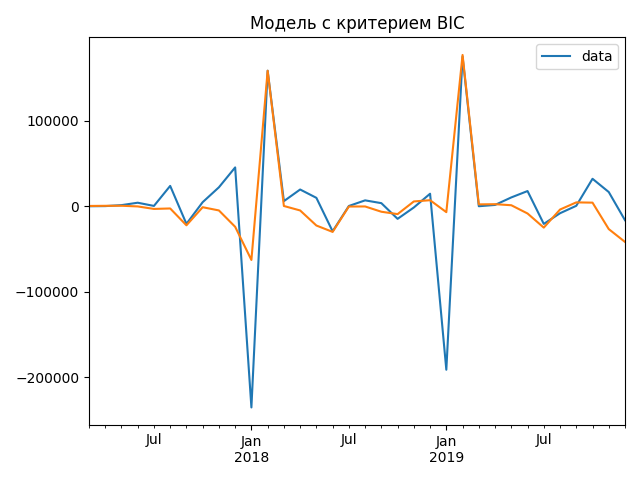
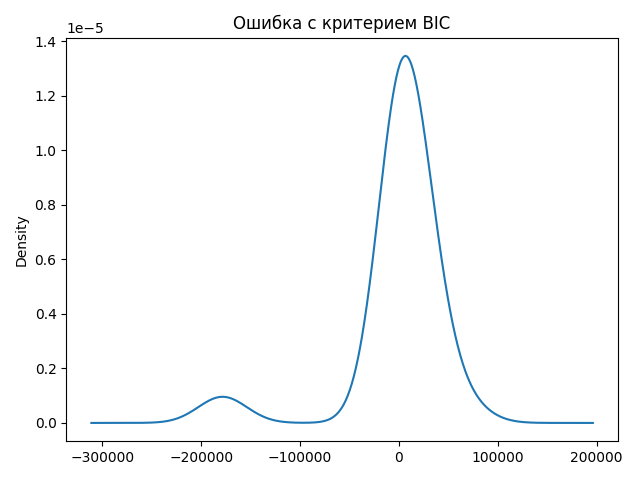
def subtask8():  
 print(**"Start subtask #8"**)  
 data = getData()  
 dates = pd.date\_range(**'2017-01-01'**, periods=36, freq=**'MS'**)  
 df = pd.DataFrame({**"data"**: data}, index=pd.DatetimeIndex(dates))  
  
 decompose = seasonal\_decompose(df,  
 model=**"additive"**,  
 extrapolate\_trend=**"freq"**)  
 do\_adfuller(df)  
 diff\_df = diff(df)  
 do\_adfuller(diff\_df)  
 diff\_df.plot(label=**"дифференцированный ряд"**)  
 plt.title(**"Дифференцированный ряд"**)  
 plt.legend()  
 plt.show()  
  
 plot\_acf(diff\_df)  
 plot\_pacf(diff\_df)  
 plt.legend()  
 plt.show()  
  
 import itertools  
  
 p = q = range(0, 3)  
 pdq = list(itertools.product(p, q))  
 for i in range(len(pdq)):  
 pdq[i] = (pdq[i][0], 0, pdq[i][1])  
 combs = {}  
 aics = []  
  
 for combination in pdq:  
 try:  
 model = SARIMAX(diff\_df, order=combination, seasonal=False)  
 model = model.fit()  
 combs.update({model.aic: combination})  
 aics.append(model.aic)  
 except:  
 continue  
 best\_aic = min(aics)  
 model = SARIMAX(diff\_df, order=combs[best\_aic], seasonal=False)  
 model = model.fit()  
 model.resid.plot(kind=**'kde'**)  
 plt.show()  
  
 print(**"############################################"**)  
 print(**"Оптимальные коэффициенты: "**, combs[best\_aic])  
 print(**"Ошибка полученного решения:"**, (abs(model.resid)).sum() / len(data))  
 print(**"End subtask #8"**)

В результате запуска перебора были получены следуюшие коэффициенты коэффициенты p, d, q — (0, 0, 0). Ошибка полученного решения: 22854.38888888889

Однако значения, предсказываемые моделью выглядят плохо, поэтому выполним дифференцирование ещё раз. После двух дифференциаций модель описывает данные намного лучше. Полученные коэффициенты p, d, q — (0, 0, 1), ошибка полученного решения: 20752.286591220254

1. **Какая модель лучше описывает временной ряд? Использовать байесовский информационный критерий (*Bayesian information criterion – BIC*) – он же критерий Шварца (*Schwarz criterion – SC*), а также критерия Акаике *AIC.***

На предыдущем шаге мы уже вычислили оптимальные коэффициенты по критерию AIC. Теперь выполним перебор с минимизацией по критерию BIC.

****В результате построения модели по критерию BIC получены коэффициенты p, d, q — (0, 0, 1), ошибка полученного решения — 20752.286591220254

Вывод: построение модели по обоим критериям даёт одинаковый уровень точности.

# Задание № 2. Метод главных компонент

1. **Построить парные диаграммы рассеяния и биплот, определить парные коэффициенты корреляции.**

Построим диаграммы рассеяния с помощью библиотеки matplotlib

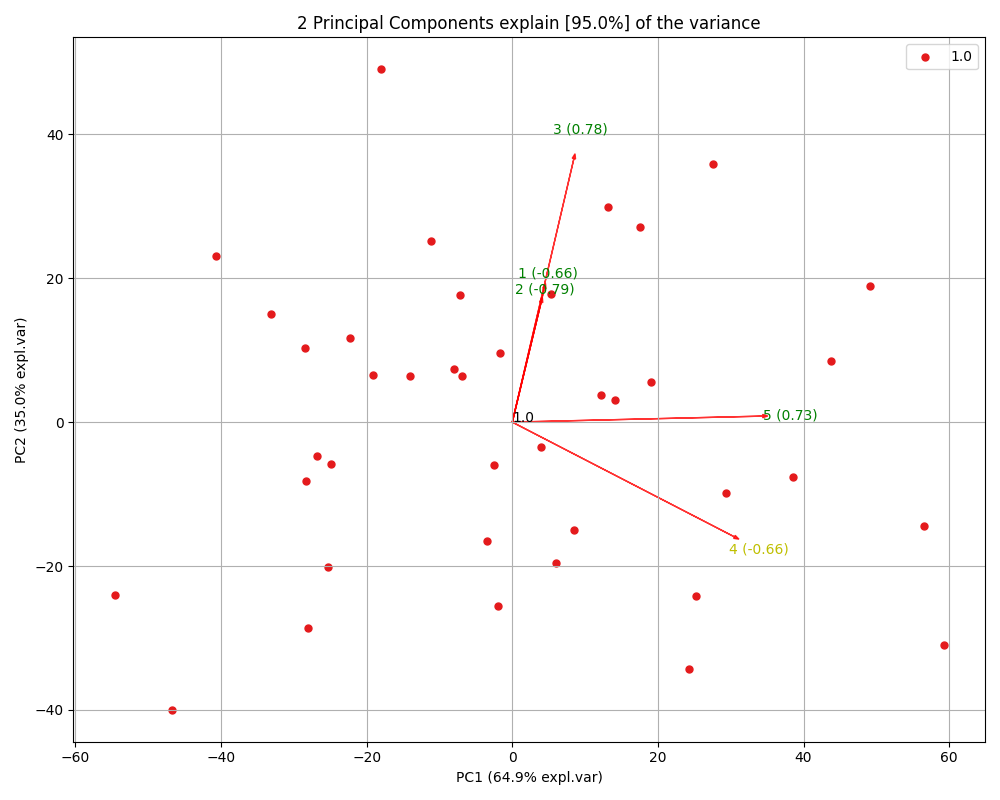
def build\_scattering\_plots(data, labels):  
 N = len(data)  
 for i in range(N):  
 for j in range(i, N):  
 if i == j:  
 continue  
 plt.title(**f"x**{i + 1} **и x**{j + 1}**"**)  
 plt.scatter(data[i], data[j])  
 plt.xlabel(labels[i])  
 plt.ylabel(labels[j])  
 plt.show()  
  
  
def subtask1():  
 build\_scattering\_plots(data, labels)

|  |  |
| --- | --- |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Анализ диаграмм рассеяния показывает сильную корреляцию между признаками x1 и x2, x1 и x3, x2 и x3. Также наблюдается некоторая корреляция между признаками x4 и x5.

Биплот построим с помощью библиотеки pca:

def build\_biplot(data):  
 model = pca.pca(n\_components=0.95)  
 model.fit\_transform(data.to\_numpy())  
 model.biplot(n\_feat=5)  
 plt.show()

Далее вычислим парные коэффициенты корреляции, чтобы подтвердить предположение о наличии корреляции:

def calculate\_corr\_coefs(data):  
 N = len(data)  
 result = [[0.0 for \_ in range(N)] for \_ in range(N)]  
 for i in range(N):  
 for j in range(N):  
 result[i][j] = np.corrcoef(data[i], data[j])[0, 1]  
 return result

corr\_coefs = calculate\_corr\_coefs(global\_data.to\_numpy().transpose())  
N = 5  
for i in range(N):  
 for j in range(i, N):  
 if i == j:  
 continue  
 print(**f"Коэффициент корреляции x**{i + 1} **и x**{j + 1}**:** {corr\_coefs[j][i]}**"**)

В результате получаем

Коэффициент корреляции x1 и x2: 0.9992547789008499  
Коэффициент корреляции x1 и x3: 0.9989454161906375  
Коэффициент корреляции x1 и x4: -0.05425326063366728  
Коэффициент корреляции x1 и x5: 0.3236568969936022  
Коэффициент корреляции x2 и x3: 0.9990116836927283  
Коэффициент корреляции x2 и x4: -0.05300994858226912  
Коэффициент корреляции x2 и x5: 0.32496767120264264  
Коэффициент корреляции x3 и x4: -0.06144939923066958  
Коэффициент корреляции x3 и x5: 0.31704325125871646  
Коэффициент корреляции x4 и x5: 0.9269046628142131

Вычисленные коэффициенты корреляции подтверждают наличие корреляции между признаками x1 и x2, x1 и x3, x2 и x3. Также подтвердилась корреляция x4 и x5.

1. **Выполнить центрирование и нормирование данных.**

Выполним центрирование исходных данных вычитанием из каждого значения среднего данного признака

def subtask2():  
 print(**"Start subtask #2"**)  
 centred\_data = global\_data - global\_data.mean()  
 print(centred\_data)  
 norm\_data = centred\_data / centred\_data.std()  
 print(norm\_data)  
 print(**"End subtask #2"**)

В результате получаем центрированные и нормализованные данные

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | x1 | x2 | x3 | x4 | x5 |
| 1 | 0.336649 | 0.352323 | 0.281498 | 0.435523 | 0.490601 |
| 2 | -0.483017 | -0.561106 | -0.552570 | -0.746428 | -0.876248 |
| 3 | 0.687934 | 0.743792 | 0.758108 | -0.540871 | -0.241639 |
| 4 | -0.834303 | -0.822086 | -0.790875 | 0.178577 | -0.144007 |
| 5 | 1.507600 | 1.526731 | 1.532600 | -0.078369 | 0.490601 |
| 6 | 0.922124 | 0.874282 | 0.877261 | -0.129758 | 0.197705 |
| 7 | 1.975981 | 1.918200 | 1.949634 | 0.281356 | 1.027577 |
| 8 | 0.453744 | 0.482812 | 0.400651 | -0.232536 | -0.046375 |
| 9 | -1.185588 | -1.213555 | -1.207909 | -0.489482 | -0.925064 |
| 10 | 0.336649 | 0.352323 | 0.341074 | -1.363098 | -1.169144 |
| 11 | 0.570839 | 0.613302 | 0.638956 | -1.774212 | -1.413224 |
| 12 | -1.653969 | -1.605025 | -1.624943 | -0.438093 | -1.022696 |
| 13 | 0.102458 | 0.091343 | 0.043193 | 1.411918 | 1.369289 |
| 14 | 2.093076 | 2.048690 | 2.068787 | -1.465877 | -0.583351 |
| 15 | 1.390505 | 1.396241 | 1.473024 | 0.127188 | 0.637049 |
| 16 | 0.336649 | 0.352323 | 0.281498 | 0.332745 | 0.441785 |
| 17 | -0.483017 | -0.561106 | -0.492994 | -0.797817 | -0.973880 |
| 18 | -0.717208 | -0.691596 | -0.671723 | -0.797817 | -1.022696 |
| 19 | 1.039219 | 1.004772 | 1.055990 | -0.797817 | -0.388087 |
| 20 | -0.131732 | -0.169637 | -0.135536 | 1.154972 | 1.027577 |
| 21 | 0.453744 | 0.482812 | 0.460227 | 0.538302 | 0.685865 |
| 22 | -0.248827 | -0.300127 | -0.314265 | 0.024410 | -0.095191 |
| 23 | -0.834303 | -0.822086 | -0.850452 | 1.257751 | 0.881129 |
| 24 | -2.356540 | -2.387964 | -2.339859 | -0.849206 | -1.706120 |
| 25 | -1.771064 | -1.735515 | -1.684520 | -1.414488 | -1.950200 |
| 26 | 0.219553 | 0.221833 | 0.221922 | -0.335315 | -0.241639 |
| 27 | 0.102458 | 0.091343 | 0.102769 | -0.746428 | -0.680983 |
| 28 | 0.219553 | 0.221833 | 0.281498 | -0.386704 | -0.290455 |
| 29 | 1.390505 | 1.396241 | 1.413448 | 1.309140 | 1.759817 |
| 30 | -1.185588 | -1.213555 | -1.207909 | 0.384134 | -0.095191 |
| 31 | 0.336649 | 0.352323 | 0.281498 | -0.951985 | -0.778615 |
| 32 | -0.131732 | -0.039147 | -0.135536 | 0.178577 | 0.148889 |
| 33 | -0.834303 | -0.822086 | -0.790875 | 2.491091 | 2.101530 |
| 34 | 0.805029 | 0.874282 | 0.877261 | 1.309140 | 1.564553 |
| 35 | -0.834303 | -0.822086 | -0.850452 | 0.538302 | 0.197705 |
| 36 | 0.219553 | 0.221833 | 0.162345 | -1.106152 | -1.022696 |
| 37 | 0.102458 | 0.091343 | 0.162345 | -0.592260 | -0.485719 |
| 38 | -0.014637 | -0.039147 | -0.075960 | 2.131367 | 2.003898 |
| 39 | -1.302683 | -1.344045 | -1.327062 | 1.411918 | 0.832313 |
| 40 | -0.600113 | -0.561106 | -0.612146 | 0.538302 | 0.295337 |

1. **Построить ковариационную матрицу**

*Ковариационная матрица –* это квадратная матрица ∑, у которой -ый элемент является ковариацией признаков :

def convert\_to\_center\_norm(data):  
 new\_data = data - data.mean()  
 new\_data = new\_data / new\_data.std()  
 return new\_data  
  
  
def subtask3():  
 print(**"Start subtask #3"**)  
 center\_norm\_data = convert\_to\_center\_norm(global\_data)  
 cov\_m = center\_norm\_data.cov()  
 print(cov\_m)  
 print(**"End subtask #3"**)

В результате получаем следующую ковариационную матрицу:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | x1 | x2 | x3 | x4 | x5 |
| 1 | 1.000000 | 0.999255 | 0.998945 | -0.054253 | 0.323657 |
| 2 | 0.999255 | 1.000000 | 0.999012 | -0.053010 | 0.324968 |
| 3 | 0.998945 | 0.999012 | 1.000000 | -0.061449 | 0.317043 |
| 4 | -0.054253 | -0.053010 | -0.061449 | 1.000000 | 0.926905 |
| 5 | 0.323657 | 0.324968 | 0.317043 | 0.926905 | 1.000000 |

1. **Найти собственные значения ковариационной матрицы**

Для нахождения собственных значений матрицы решается характеристическое уравнение:

Мы воспользуемся библиотекой *numpy* и её функцией *linalg*:

def get\_cov\_matrix(data):  
 center\_norm\_data = convert\_to\_center\_norm(global\_data)  
 cov\_m = center\_norm\_data.cov()  
 return cov\_m  
  
  
def subtask4():  
 print(**"Start subtask #4"**)  
 cov\_matrix = get\_cov\_matrix(global\_data)  
 e\_values, e\_vectors = np.linalg.eig(cov\_matrix.to\_numpy())  
 indexes = np.argsort(-e\_values)  
 print(**"Собственные значения: "**, e\_values)  
 e\_vectors = e\_vectors[indexes]  
 print(**"Собственные векторы: "**, e\_vectors)  
 print(**"End subtask #4"**)

Полученные собственные значения

[3.15 1.84 1.09371082e-03 7.42747318e-04 2.01640017e-04]

1. **Найти ортогональную матрицу собственных векторов ковариационной матрицы.**

Для каждого собственного значения находим собственный вектор с помощью СЛУ:

При этом должно соблюдаться условие:

Т. е. требуется нормировать собственные вектора:

Функция *linalg* библиотеки numpy уже возвращает собственные нормированные векторы, нам надо их отсортировать по собственным значениям:

def get\_cov\_matrix(data):  
 center\_norm\_data = convert\_to\_center\_norm(global\_data)  
 cov\_m = center\_norm\_data.cov()  
 return cov\_m  
  
  
def subtask4():  
 print(**"Start subtask #4"**)  
 cov\_matrix = get\_cov\_matrix(global\_data)  
 e\_values, e\_vectors = np.linalg.eig(cov\_matrix.to\_numpy())  
 indexes = np.argsort(-e\_values)  
 print(**"Собственные значения: "**, e\_values)  
 e\_vectors = e\_vectors[indexes]  
 print(**"Собственные векторы: "**, e\_vectors)  
 print(**"End subtask #4"**)

Получаем:

Собственные векторы:

[[-0.55248472 -0.14102981 -0.51593687 0.63849562 0.03168149]  
 [-0.55264375 -0.14008931 -0.30835778 -0.75252748 -0.11652355]  
 [-0.55167539 -0.1461493 0.78914897 0.13701301 -0.18101377]  
 [-0.0776691 0.72934419 -0.08369355 0.05960091 -0.67191346]  
 [-0.2810112 0.63810894 0.09467797 -0.060835 0.70794411]]

# Задание № 3. Кластерный анализ

Алгомеративные методы кластеризации основаны на построении дерева иерархии от листьев к стволу путём объединения более мелких кластеров.

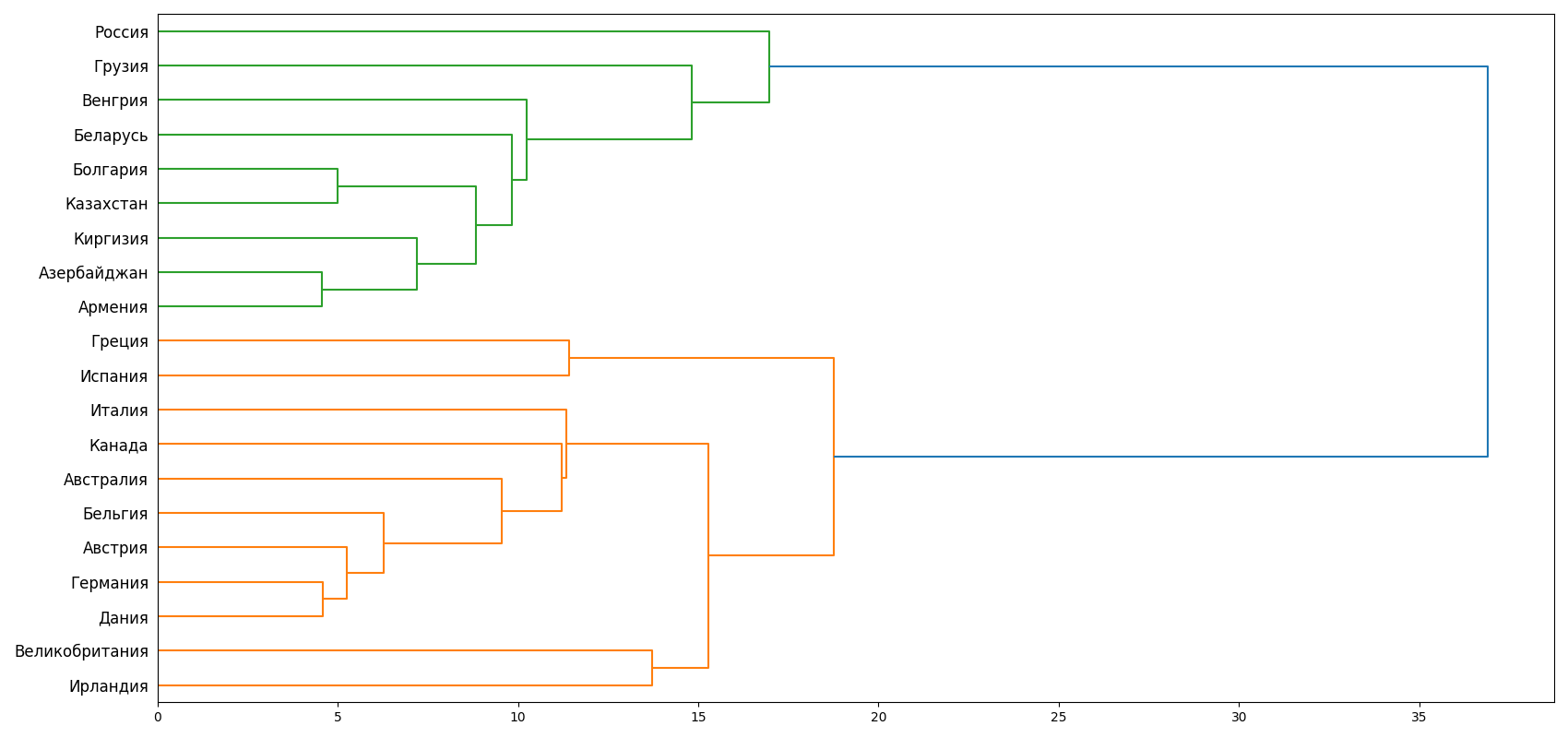
1. **Провести классификацию по иерархическому агломеративному алгоритму с использованием евклидова расстояния и метода «ближнего соседа». Построить дендограмму.**

Евклидово расстояние:

Метод ближнего соседа состоит в том, что на каждом этапе алгоритма мы присваиваем объекту класс ближайшего соседа. Метод дальнего соседа – класс самого удаленного соседа.

Чтобы выполнить классификацию и построение дендограмм воспользуемся библиотекой scipy (метод hierarchy) и библиотекой matplotlib.

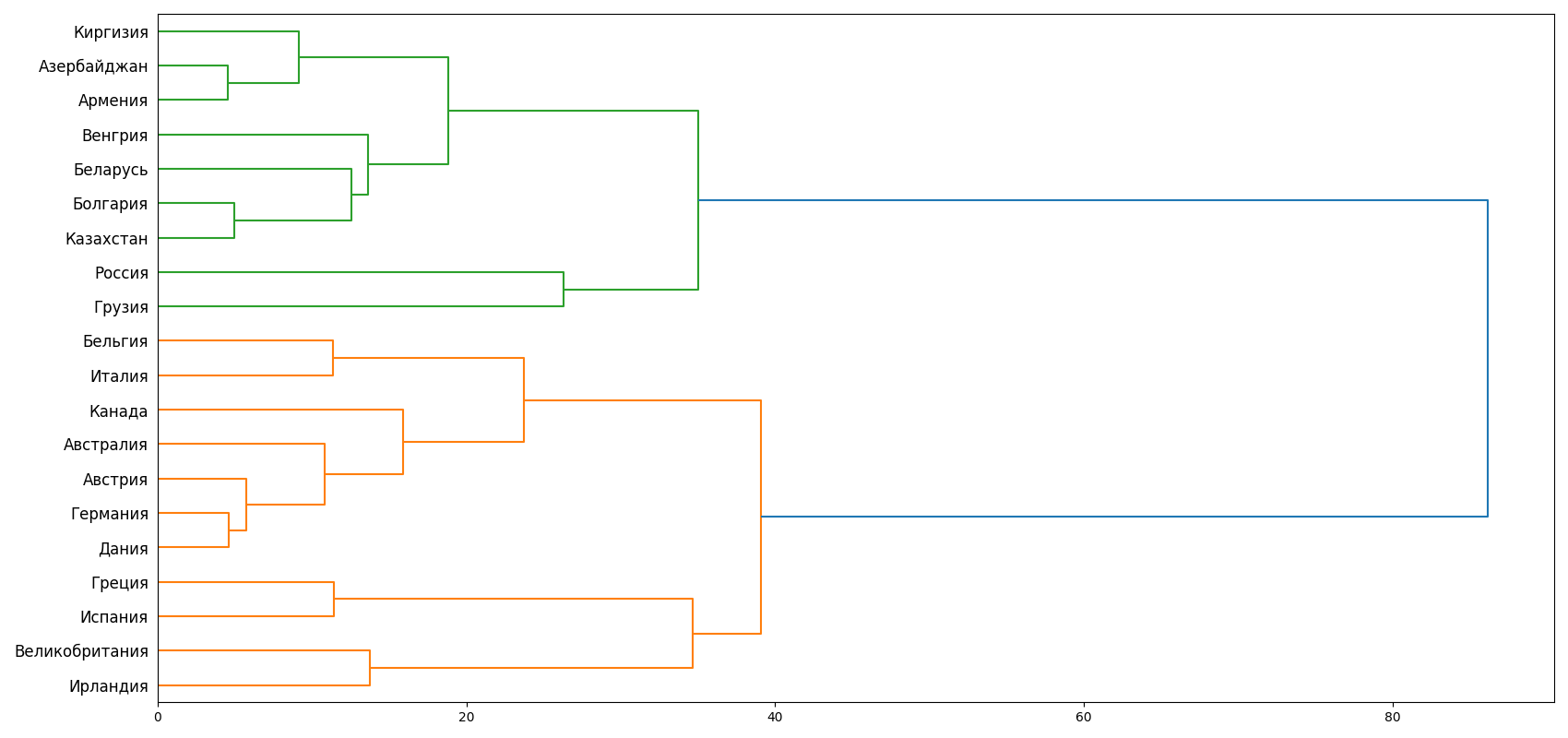
def build\_dendogram(clusterd\_date, labels):  
 plt.figure()  
 dendrogram(clusterd\_date, orientation=**'right'**, labels=labels, distance\_sort=**'descending'**, show\_leaf\_counts=True)  
 plt.show()  
  
  
def euclidean\_nearest\_neighbour\_clustering(data, labels):  
 clusterd\_date = linkage(data, **'single'**, metric=**'euclidean'**)  
 build\_dendogram(clusterd\_date, labels)  
  
  
def subtask1():  
 euclidean\_nearest\_neighbour\_clustering(data, countries)

В результате получаем следующую дендограмму:

1. **Провести классификацию по иерархическому агломеративному алгоритму с использованием** **евклидова расстояния и метода «дальнего соседа». Построить дендограмму.**

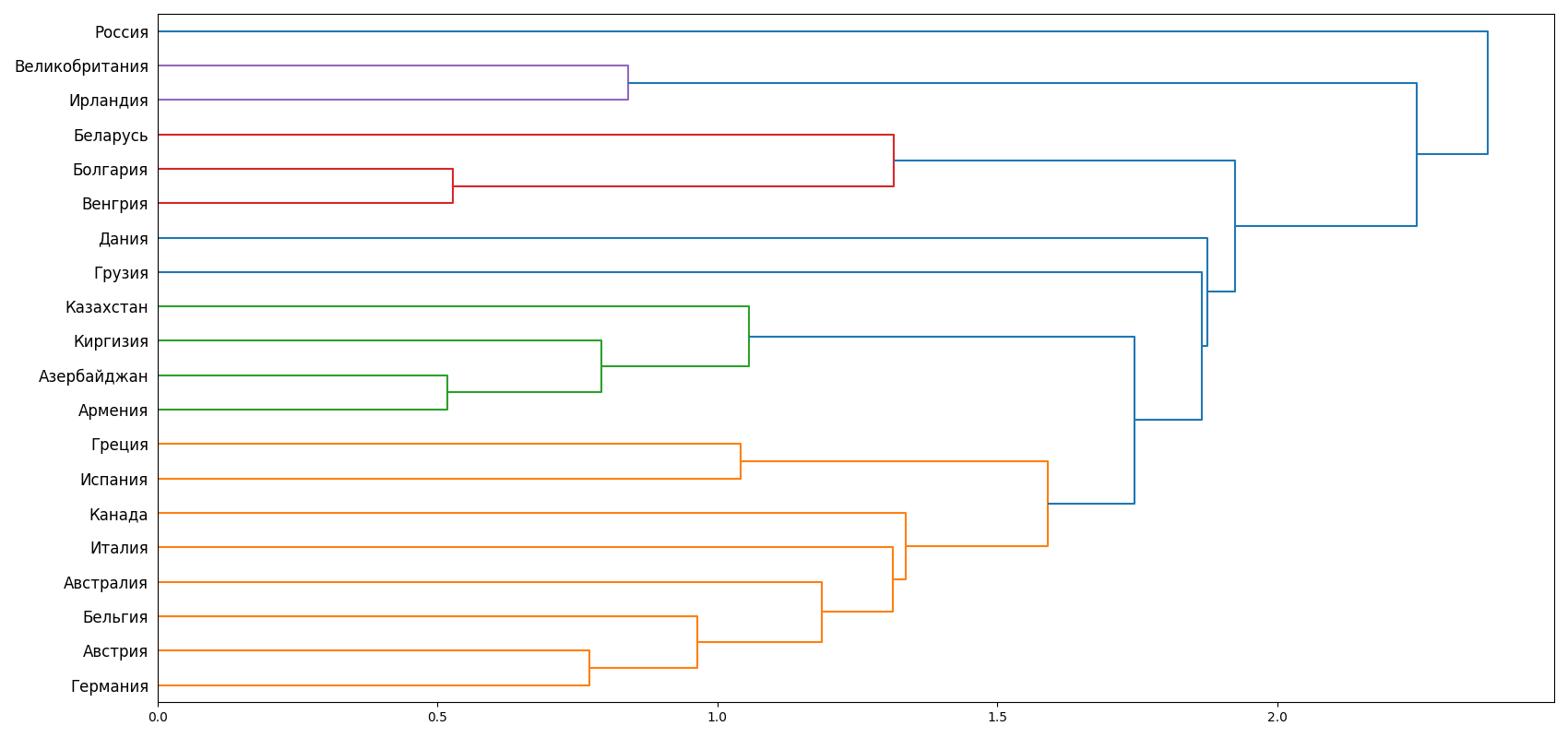
Метод дальнего соседа — правило объединения, подразумевающее, что новый объект присоединяется к тому кластеру, самый далекий элемент которого находится ближе к новому объекту, чем самые далекие элементы других кластеров. Это правило является противоположным предыдущему и более жестким. Поэтому здесь наблюдается тенденция к выделению большего числа компактных кластеров, состоящих из наиболее похожих элементов.

def euclidean\_further\_neighbour\_clustering(data, labels):  
 clusterd\_date = linkage(data, **'complete'**, metric=**'euclidean'**)  
 build\_dendogram(clusterd\_date, labels)  
  
  
def subtask2():  
 euclidean\_further\_neighbour\_clustering(data, countries)



1. **Провести классификацию по иерархическому агломеративному алгоритму с использованием** **расстояния Махаланобиса и метода «ближнего соседа» . Построить дендограмму.**

Расстояние Махалано́биса — мера расстояния между векторами случайных величин, обобщающая понятие евклидова расстояния.

def mahalanobis\_nearest\_neighbour\_clustering(data, labels):  
 clusterd\_date = linkage(data, **'single'**, metric=**'mahalanobis'**)  
 build\_dendogram(clusterd\_date, labels)  
  
  
def subtask3():  
 mahalanobis\_nearest\_neighbour\_clustering(data, countries)