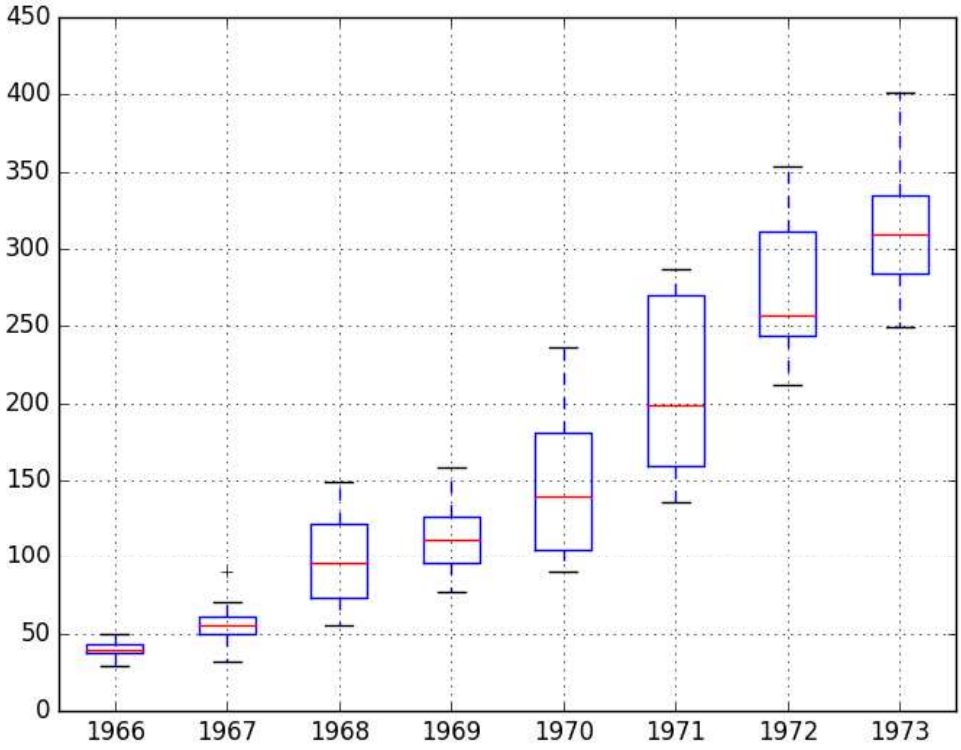

Figure 30.3: Monthly box and whisker plots of the training set for the Boston Robberies dataset.

The observations suggest that the year-to-year fluctuations may not be systematic and hard to model. They also suggest that there may be some benefit in clipping the first two years of data from modeling if it is indeed quite different. This yearly view of the data is an interesting avenue and could be pursued further by looking at summary statistics from year-to-year and changes in summary stats from year-to-year. Next, we can start looking at predictive models of the series.

30.6 ARIMA Models

In this section, we will develop Autoregressive Integrated Moving Average, or ARIMA, models for the problem. We will approach this in four steps:

1. Developing a manually configured ARIMA model.
2. Using a grid search of ARIMA to find an optimized model.
3. Analysis of forecast residual errors to evaluate any bias in the model.
4. Explore improvements to the model using power transforms.

30.6.1 Manually Configured ARIMA

Nonseasonal `ARIMA(p,d,q)` requires three parameters and is traditionally configured manually. Analysis of the time series data assumes that we are working with a stationary time series. The time series is almost certainly non-stationary. We can make it stationary by first differencing the series and using a statistical test to confirm that the result is stationary. The example below creates a stationary version of the series and saves it to file `stationary.csv`.

```
# statistical test for the stationarity of the time series
from pandas import Series
from statsmodels.tsa.stattools import adfuller

# create a differenced time series
def difference(dataset):
  diff = list()
  for i in range(1, len(dataset)):
    value = dataset[i] - dataset[i - 1]
    diff.append(value)
  return Series(diff)

series = Series.from_csv('dataset.csv')
X = series.values
# difference data
stationary = difference(X)
stationary.index = series.index[1:]
# check if stationary
result = adfuller(stationary)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
  print('\t%s: %.3f' % (key, value))
# save
stationary.to_csv('stationary.csv')
```

Listing 30.14: Create a stationary version of the Boston Robberies dataset.

Running the example outputs the result of a statistical significance test of whether the 1-lag differenced series is stationary. Specifically, the augmented Dickey-Fuller test. The results show that the test statistic value -3.980946 is smaller than the critical value at 5% of -2.893. This suggests that we can reject the null hypothesis with a significance level of less than 5% (i.e. a low probability that the result is a statistical fluke). Rejecting the null hypothesis means that the process has no unit root, and in turn that the 1-lag differenced time series is stationary or does not have time-dependent structure.

```
ADF Statistic: -3.980946
p-value: 0.001514
Critical Values:
  1%: -3.503
  5%: -2.893
  10%: -2.584
```

Listing 30.15: Example output of stationarity test on the differenced Boston Robberies dataset.

This suggests that at least one level of differencing is required. The `d` parameter in our ARIMA model should at least be a value of 1. The next step is to select the lag values for

the Autoregression (AR) and Moving Average (MA) parameters, p and q respectively. We can do this by reviewing Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots. The example below creates ACF and PACF plots for the series.

```
# ACF and PACF plots of time series
from pandas import Series
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from matplotlib import pyplot
series = Series.from_csv('dataset.csv')
pyplot.figure()
pyplot.subplot(211)
plot_acf(series, ax=pyplot.gca())
pyplot.subplot(212)
plot_pacf(series, ax=pyplot.gca())
pyplot.show()
```

Listing 30.16: Create ACF and PACF plots of the Boston Robberies dataset.

Run the example and review the plots for insights into how to set the p and q variables for the ARIMA model. Below are some observations from the plots.

- The ACF shows a significant lag for 10-11 months.
- The PACF shows a significant lag for perhaps 2 months.
- Both the ACF and PACF show a drop-off at the same point, perhaps suggesting a mix of AR and MA.

A good starting point for the p and q values is 1 or 2.

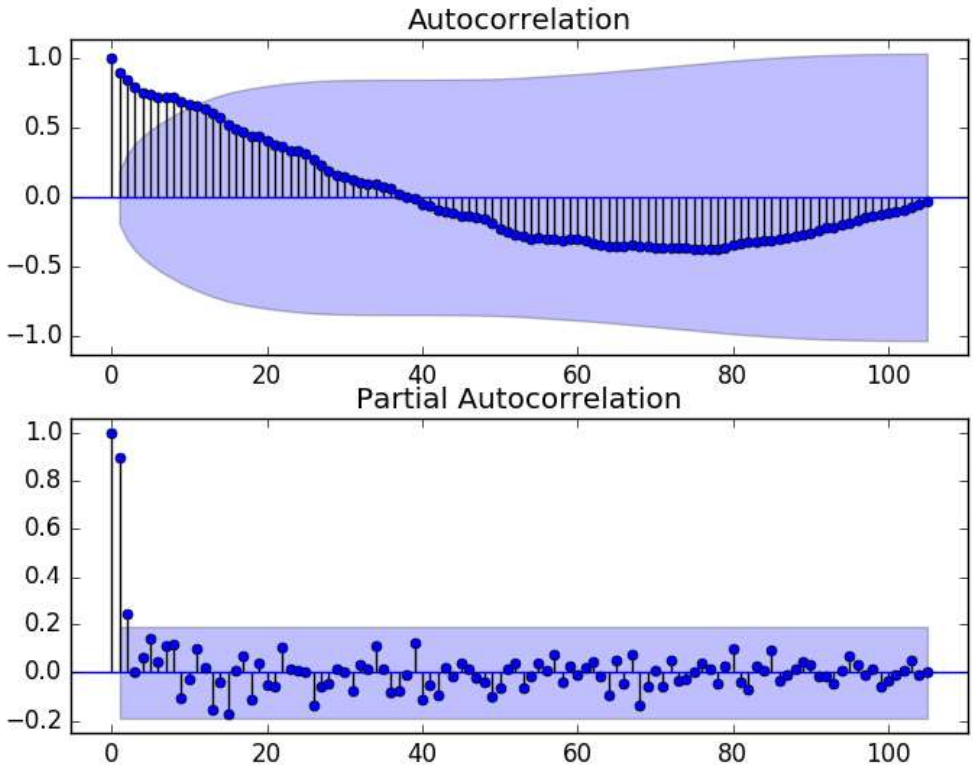

Figure 30.4: ACF and PACF plots of the training set for the Boston Robberies dataset.

This quick analysis suggests an `ARIMA(11,1,2)` on the raw data may be a good starting point. Experimentation shows that this configuration of ARIMA does not converge and results in errors by the underlying library, as do similarly large AR values. Some experimentation shows that the model does not appear to be stable, with non-zero AR and MA orders defined at the same time. The model can be simplified to `ARIMA(0,1,2)`. The example below demonstrates the performance of this ARIMA model on the test harness.

```
# evaluate manually configured ARIMA model
from pandas import Series
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from math import sqrt
# load data
series = Series.from_csv('dataset.csv')
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
```

```
  # predict
  model = ARIMA(history, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 30.17: Manual ARIMA model for the Boston Robberies dataset.

Running this example results in an RMSE of 49.821, which is lower than the persistence model.

```
...
>Predicted=280.614, Expected=287
>Predicted=302.079, Expected=355
>Predicted=340.210, Expected=460
>Predicted=405.172, Expected=364
>Predicted=333.755, Expected=487
RMSE: 49.821
```

Listing 30.18: Example output of a manual ARIMA model for the Boston Robberies dataset.

This is a good start, but we may be able to get improved results with a better configured ARIMA model.

30.6.2 Grid Search ARIMA Hyperparameters

Many ARIMA configurations are unstable on this dataset, but there may be other hyperparameters that result in a well-performing model. In this section, we will search values of `p`, `d`, and `q` for combinations that do not result in error, and find the combination that results in the best performance. We will use a grid search to explore all combinations in a subset of integer values. Specifically, we will search all combinations of the following parameters:

- `p`: 0 to 12.
- `d`: 0 to 3.
- `q`: 0 to 12.

This is $(13 \times 4 \times 13)$, or 676, runs of the test harness and will take some time to execute. The complete worked example with the grid search version of the test harness is listed below.

```
# grid search ARIMA parameters for time series
import warnings
from pandas import Series
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
```

```
# evaluate an ARIMA model for a given order (p,d,q) and return RMSE
def evaluate_arima_model(X, arima_order):
  # prepare training dataset
  X = X.astype('float32')
  train_size = int(len(X) * 0.50)
  train, test = X[0:train_size], X[train_size:]
  history = [x for x in train]
  # make predictions
  predictions = list()
  for t in range(len(test)):
    model = ARIMA(history, order=arima_order)
    model_fit = model.fit(disp=0)
    yhat = model_fit.forecast()[0]
    predictions.append(yhat)
    history.append(test[t])
  # calculate out of sample error
  rmse = sqrt(mean_squared_error(test, predictions))
  return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
  dataset = dataset.astype('float32')
  best_score, best_cfg = float("inf"), None
  for p in p_values:
    for d in d_values:
      for q in q_values:
        order = (p,d,q)
        try:
          rmse = evaluate_arima_model(dataset, order)
          if rmse < best_score:
            best_score, best_cfg = rmse, order
          print('ARIMA%s RMSE=%.3f' % (order,rmse))
        except:
          continue
  print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
series = Series.from_csv('dataset.csv')
# evaluate parameters
p_values = range(0,13)
d_values = range(0, 4)
q_values = range(0, 13)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 30.19: Grid Search ARIMA models for the Boston Robberies dataset.

Running the example runs through all combinations and reports the results on those that converge without error. The example takes a little less than 2 hours to run on modern hardware. The results show that the best configuration discovered was `ARIMA(0,1,2)`; coincidently, that was demonstrated in the previous section.

```
...
ARIMA(6, 1, 0) RMSE=52.437
ARIMA(6, 2, 0) RMSE=58.307
ARIMA(7, 1, 0) RMSE=51.104
```

```
ARIMA(7, 1, 1) RMSE=52.340
ARIMA(8, 1, 0) RMSE=51.759
Best ARIMA(0, 1, 2) RMSE=49.821
```

Listing 30.20: Example output of grid searching ARIMA models for the Boston Robberies dataset.

30.6.3 Review Residual Errors

A good final check of a model is to review residual forecast errors. Ideally, the distribution of residual errors should be a Gaussian with a zero mean. We can check this by plotting the residuals with a histogram and density plots. The example below calculates the residual errors for predictions on the test set and creates these density plots.

```
# plot residual errors for ARIMA model
from pandas import Series
from pandas import DataFrame
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from math import sqrt
from matplotlib import pyplot
# load data
series = Series.from_csv('dataset.csv')
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # predict
  model = ARIMA(history, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
pyplot.figure()
pyplot.subplot(211)
residuals.hist(ax=pyplot.gca())
pyplot.subplot(212)
residuals.plot(kind='kde', ax=pyplot.gca())
pyplot.show()
```

Listing 30.21: Plot density of residual errors from ARIMA model on the Boston Robberies dataset.

Running the example creates the two plots. The graphs suggest a Gaussian-like distribution

with a longer right tail. This is perhaps a sign that the predictions are biased, and in this case that perhaps a power-based transform of the raw data before modeling might be useful.

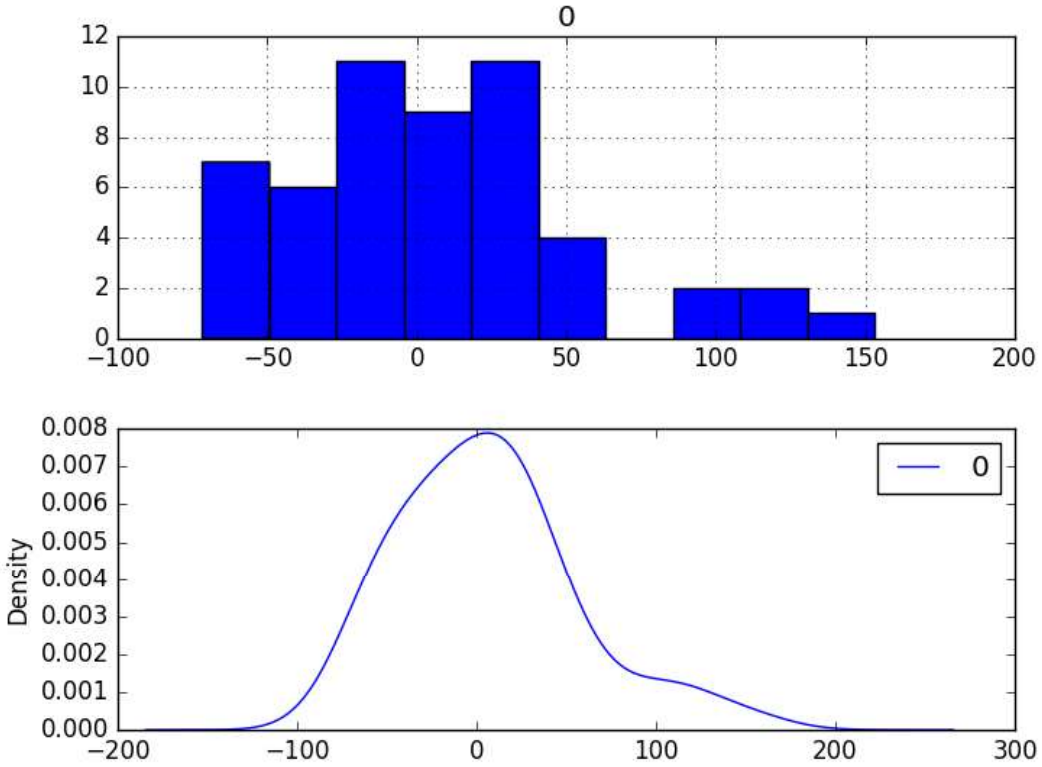

Figure 30.5: Density plots of the residual errors from ARIMA models on the Boston Robberies dataset.

It is also a good idea to check the time series of the residual errors for any type of autocorrelation. If present, it would suggest that the model has more opportunity to model the temporal structure in the data. The example below re-calculates the residual errors and creates ACF and PACF plots to check for any significant autocorrelation.

```
# ACF and PACF plots of forecast residual errors
from pandas import Series
from pandas import DataFrame
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from math import sqrt
from matplotlib import pyplot
# load data
series = Series.from_csv('dataset.csv')
# prepare data
X = series.values
X = X.astype('float32')
```

```
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # predict
  model = ARIMA(history, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
pyplot.figure()
pyplot.subplot(211)
plot_acf(residuals, ax=pyplot.gca())
pyplot.subplot(212)
plot_pacf(residuals, ax=pyplot.gca())
pyplot.show()
```

Listing 30.22: ACF and PACF plots of residual errors from ARIMA model on the Boston Robberies dataset.

The results suggest that what little autocorrelation is present in the time series has been captured by the model.

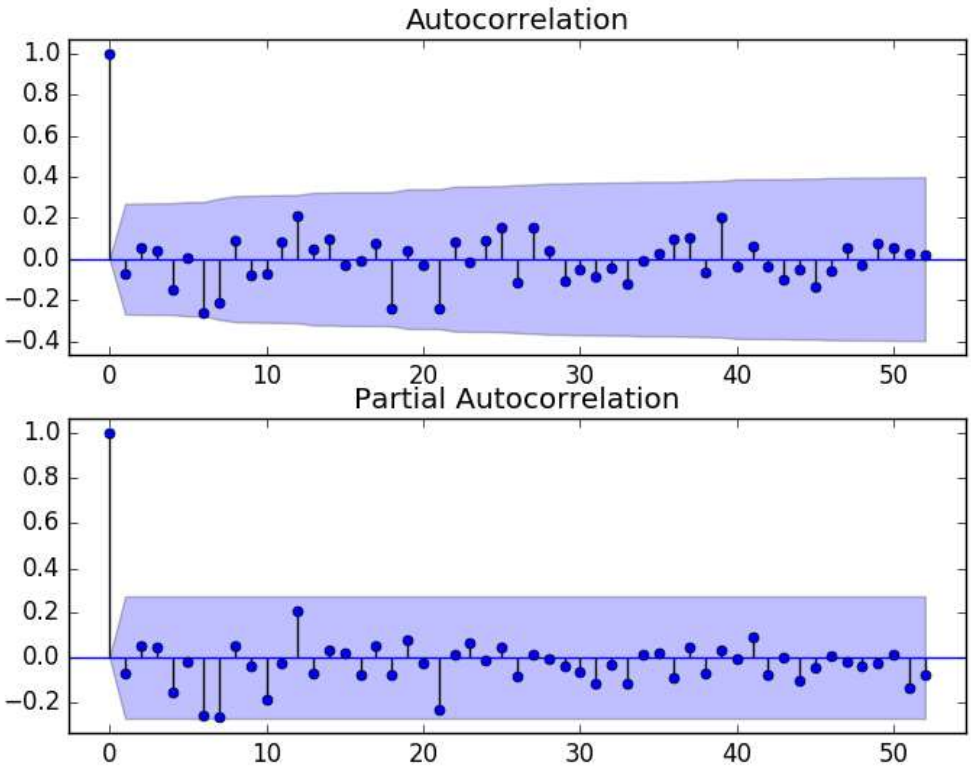

Figure 30.6: ACF and PACF plots of the residual errors from ARIMA models on the Boston Robberies dataset.

30.6.4 Box-Cox Transformed Dataset

The Box-Cox transform is a method that is able to evaluate a suite of power transforms, including, but not limited to, log, square root, and reciprocal transforms of the data. The example below performs a log transform of the data and generates some plots to review the effect on the time series.

```
# plots of box-cox transformed dataset
from pandas import Series
from pandas import DataFrame
from scipy.stats import boxcox
from matplotlib import pyplot
from statsmodels.graphics.gofplots import qqplot
series = Series.from_csv('dataset.csv')
X = series.values
transformed, lam = boxcox(X)
print('Lambda: %f' % lam)
pyplot.figure(1)
# line plot
pyplot.subplot(311)
pyplot.plot(transformed)
# histogram
```