pylookml

Release 3.0.0

Sep 16, 2021

Contents

```
view + f'''dimension: {{dim}} {
              type: number
              sql: {{dim_sql}} ;;
            }'''
```

PyLookML allows scripting of LookML in python. It leverages the lkml parser to interpret raw lookml files then adds an object oriented syntax and helpful integrations to boost your productivity. View the source code or log an issue here.

Note: pyLookML 3.0.0, a milestone release, is now available on pip. See the changelog for details.

CHAPTER 1

Why

Sometimes usecases demand automation, where you want rules to govern the rules.

- EAV data / frequently changing custom fields (see EAV)
- Nested data
- Applying complex patterns repeatably
- Create LookML based on API response (such as autotune)
- Bulk conversion of old reports

CHAPTER 2

Quickstart

Install pylookml package via pip

```
pip install lookml
```

Make a github access token

Fetch a viewFile from Github and print one of its dimensions

```
import lookml
proj = lookml.Project(
       repo= "llooker/pyLookMLExample",
       access_token="your_github_access_token",
       #Optional args for the deploy URL (for deploying directly to prodcution mode)
      ,looker_host="https://mylooker.looker.com/"
      ,looker_project_name="my_project"
)
viewFile = proj.file('01_order_items.view.lkml')
orderItems = viewFile.views.order_items
print(orderItems.id)
```

Or do the same thing from any other git service (as long as you have SSH git access on the machine pyLookML is running on):

```
     self.proj = lookml.Project(
                git_url='git@bitbucket.org:myorg/russ_sandbox.git'
                #Optional args for the deploy URL (for deploying directly to prodcution mode)
               ,looker_host="https://mylooker.looker.com/"
               ,looker_project_name="my_project"
     )
```

This works for bitbucket, gitlab, or private git servers.

How to reference objects The taxonomy is basically as follows project>file>'views'>viewname>fieldname>property project>file>'explores'>explorename>joinname>property

```
myProject = lookml.Project(
     repo= "llooker/russ_sandbox",
     access_token="your_github_access_token",
)
#Use a dot operator syntax:
myProject.file('order_items.view.lkml').views.order_items.id.primary_key.value

#Use a dictionary like syntax:
myProject['order_items.view.lkml']['views']['order_items']['id'].primary_key.value
```

Get all the way down to property values in one line of code

```
lookml.Project(**config['project1'])['order_items.view.lkml']['views']['order_items'][
↪'id'].primary_key.value
```

Looping over stuff

```
dimension: id {
    type: string
    sql: ${TABLE}.id ;;
    tags: ["a","b","c"]
}
```

```
for tag in order_items.id.tags:
    print(tag)
>>> 'a'
>>> 'b'
>>> 'c'
```

Updating things

The + operator in pyLookML is significant, it allows us to add a string of LookML to our object like so. Also notice the way we change the primary key paramter.

```
1  order_items = lookml.View('order_items')
2  order_items + '''
3      dimension: id {
4          type: string
5          sql: ${TABLE}.id ;;
6          }
7  '''
8  order_items.id.primary_key = 'yes'
9  print(order_items)
```

after your object is updated, you need to save it back to github, and optionally hit the looker deploy URL

```
1  newFile = lookml.File(order_items)
2  #the put method, creates or overwrites
3  myProject.put(newFile)
4  #optionally hitting the Looker deploy URL (requires that you set your instance URL␣
   ↪on project creation)
5  myProject.deploy()
```

CHAPTER 3

Build from a developer version

Step 1) Create a virtual env from a clean python and install the dependencies

```
which python3 #(this is generally the best interpreter use as the startingpoint)
#Output: /Library/Frameworks/Python.framework/Versions/3.8/bin/python3
mkdir lookml_test
cd lookml_test
virtualenv -p /Library/Frameworks/Python.framework/Versions/3.8/bin/python3 lookml_
↪test_env
source lookml_test_env/bin/activate
pip install pygithub
pip install lkml
```

Step 2) go to github and look for the specific commit you'd like to build and replace it in the following command after the @ sign

```
pip install git+https://github.com/llooker/lookml.
↪git@04dbd05dd3f37a7fa624501a370df52af26bb5fc
```

3.1 Introduction

Visit Home for the quickstart guide

Video tutorials coming soon!

3.2 Cookbook / Examples

3.2.1 Basic Recipes

- connect to your github project

```
import lookml
proj = lookml.Project(
      repo= "llooker/russ_sandbox",
      access_token="your_github_access_token",
)
```

Note: Project() will dispatch the correct project subclass ProjectGithub or ProjectSSH depending on the args provided

Or do the same thing from any other git service (as long as you have SSH git access on the machine pyLookML is running on):

```
    proj = lookml.ProjectSSH(
              git_url='git@bitbucket.org:myorg/russ_sandbox.git'
              #Optional args for the deploy URL (for deploying directly to prodcution mode)
             ,looker_host="https://mylooker.looker.com/"
             ,looker_project_name="my_project"
    )
```

Or just connect to the local filesystem without git:

```
    proj = lookml.Project(
              path='path/to/myproject'
    )
```

- Loop over the views in a file

```
myFile = proj.file('01_order_items.view.lkml')
#Loops over 1:n views in the file
for view in myFile.views:
    print(view)
```

- create a new file in your project

```
newFile = proj.new_file('views/my_new_file.view.lkml')
newFile + 'view: new_view {}'
newFile.views.new_view + lookml.Dimension('dimension: id {}')
```

- create a new model file

```
modelFile = proj.new_file('my_new_model.model.lkml')
modelFile + 'explore: foo {}'
```

- Write your files back to github

```
viewFile = proj.file('01_order_items.view.lkml')
viewFile.views.order_items.id.addTag("hello, World!")
proj.put(viewFile)
```

- Loop over fields of a certain type

```
>>> for dim in myFile.views.order_items.dimensions():
...     print(dim.__ref__)
...
${order_items.new_dimension}
```

(continues on next page)

(continued from previous page)

```
${order_items.id}
${order_items.cpt_code_value}
${order_items.inventory_item_id}
...
>>> for meas in myFile.views.order_items.measures():
...     print(meas.__ref__)
...
${order_items.count}
${order_items.min_sale_price}
${order_items.max_sale_price}
${order_items.order_count}
>>> for flt in myFile.views.order_items.filters():
...     print(flt.__ref__)
...
${order_items.cpt_code}
${order_items.cohort_by}
${order_items.metric}
```

- check all of the children / decendants of a field

```
>>> for child in order_items.sale_price.children():
...     print(child.__refs__)
...
${min_sale_price}
${max_sale_price}
${total_sale_price}
${average_sale_price}
${median_sale_price}
${returned_total_sale_price}
${gross_margin}
${item_gross_margin_percentage}
```

- search a view for dimensions who's properties match a regex pattern (Find view fields by regex searching any parameter)

```
>>> for item in order_items.search('sql','\$\{shipped_raw\}'):
...     print(item.__ref__)
...     print(item.sql)
...
${order_items.shipping_time}
sql: datediff('day',${shipped_raw},${delivered_raw})*1.0 ;;
```

- Add a new view to an existing file

```
myNewView = lookml.View('hello_world')
myFile = proj.file('01_order_items.view.lkml')
myFile + myNewView
for view in myFile.views:
   print(view.name)
>>> 'order_items'
>>> 'hello_world'
```

- Get fields by tag, do work, remove tag

```
for field in orderItems.getFieldsByTag('x'):
   #do work
   field.removeTag('x')
```

3.2.2 Field References

```
>>> myView = View('order_items') + 'id'
>>> print(myView.id)
dimension: id {

   }
#__ref__ stands for reference
>>> print(myView.id.__ref__)
${order_items.id}
#__refs__ stands for reference short
>>> print(myView.id.__refs__)
${id}
#__refr__ stands for reference raw
>>> print(myView.id.__refr__)
order_items.id
#__refrs__ stands for reference raw short
>>> print(myView.id.__refrs__)
id
```

3.2.3 Convenience Methods

- Add a sum measure for every number dimension on a view

```
orderItems.sumAllNumDimensions()
```

- Change the name of a field and all its child references

```
>>> print(order_items2.shipping_time)

dimension: shipping_time {
  type: number
  sql: datediff('day',${shipped_raw},${delivered_raw})*1.0 ;;
}

>>> for field in order_items2.shipping_time.children():
...     print(field)

measure: average_shipping_time {
  type: average
  value_format_name: decimal_2
  sql: ${shipping_time} ;;
}
#The setName_safe method previously change_name_and_child_references, use that if␣
↪setName_safe not found
>>> order_items2.shipping_time.setName_safe('time_in_transit')
>>> print(time_in_transit)
dimension: time_in_transit {
  type: number
  sql: datediff('day',${shipped_raw},${delivered_raw})*1.0 ;;
}
>>> for field in order_items2.time_in_transit.children():
...     print(field)
measure: average_shipping_time {
  type: average
  value_format_name: decimal_2
```

(continues on next page)

(continued from previous page)

```
  sql: ${time_in_transit} ;;
}
```

- working with a local file
- Changing field names safely (The setName_safe method previously change_name_and_child_references, use that if setName_safe not found)

```
myFile = lookml.File('example.view.lkml')
   for v in myFile.views:
      for f in v.measures():
            if f.type.value == 'sum' and not f.name.endswith('_total'):
               f.name = f.setName_safe(f.name + '_total')
   #Optionally Change the location
   myFile.setFolder('pathto/other/folder')
   #Write the file
   x.write()
```

3.3 AutoGen for EAV

Warning: Setting up EAV automation can generate high code volume. Pair with a Looker architect to plan for scale. Multiple instances may be necessary at large volumes.

3.3.1 What is EAV?

EAV data is storing key / value pairs in a table. It can allow application owners to hold data for which they can't predict the columns or attributes at design time. Common examples might include customizable objects (i.e. my users can add their own fields),or scientific data with many attributes or surveys. EAV data allows flexibility, but can be notoriously difficult to perform analysis on. In this tutorial, we will show how pyLookML can be configured to create LookML for unpacking, imposing a permission structure and allowing analysis on EAV data.

An Example of a configurable user profile table

Our example will follow a site with a configurable user profile. Organizations that use the site "Orgs" can add profile fields for their members so that admins can track org specific values for each of their user accounts.

Here is the sample data we'll be using throughout. Imagine this sample data comes from a table called custom_profile_fields.

Table 1: custom_profile_fields

user_id	org_id	field_name	value	datatype
1	8	c_donation_amount	40	int
1	8	c_highest_achievement	gold badge	varchar
2	101	c_highest_achievement	silver badge	varchar
2	101	c_monthly_contribution	300	int
3	101	c_highest_achievement	bronze badge	varchar
3	101	c_monthly_contribution	350	int
4	101	c_monthly_contribution	350	int
4	101	age	32	int
5	102	c_monthly_contribution	100	int

You can see that the field name and value form the key,value relationship characteristic of EAV. Structured in a traditional table layout, we would need 4 columns to capture the 4 distinct custom fields: c_donation_amount, c_highest_achievement, c_monthly_contribution, age. And this would grow (as orgs and user accounts were added) to be much wider than is practical, or wider than the database may even allow a table to be. However for analysis, we want to create a "slice" of this table for each org, showing them just their attributes as if it were a normal table. Also notice that because the "value" column has mixed datatypes it must be a wide and neutral (typically a very wide varchar) and cast by the application when the record is read. Often by necessity you will often see the value paired with a column which tracks its type so the application can bind it to the right datatype at runtime.

Here is the LookML starting point (the script assumes that you have already created views for the relevant tables) but it will allow the ongoing programatic addition of fields. We have a usr table which tracks basic information about our user accounts eav_source (which would be pointed at public.custom_profile_fields) and usr_profile which will track the extended profile attributes from custom_profile_fields (we'll also permission the fields at the org level). The explore usr, just associates our usr table to the usr profile table which will contain the un-packed EAV values. We have also added an access filter, so that our orgs can only see thier own records.

```
connection: "snowlooker"

explore: usr {
    access_filter: {
        field: usr_profile.org_id
        user_attribute: org_id
    }
    join: usr_profile {
        type: left_outer
        relationship: one_to_one
        sql_on: ${usr.id} =  ${usr_profile.user_id} ;;
    }
}

view: usr {
    sql_table_name: public.users ;;
    dimension: email {}
    dimension: id {}
    dimension_group: created { timeframes: [raw,date,month,year] }
}

view: usr_profile {
    dimension: org_id {}
    dimension: user_id {}
}

view: eav_source {
    sql_table_name: public.custom_profile_fields ;;
    dimension: datatype { type: string }
    dimension: field_name { type: string }
    dimension: org_id { type: number }
    dimension: user_id { type: number }
    dimension: value { type: string }
}
```

Now for the automation code. First install the dependencies (FYI I highly reccoemnd using a virtual environment). We will be using the Looker SDK to run sql against the DB which will tell us what fields we need to create. And we'll install our pyLookML package as well.

```
pip install lookml, looker_sdk
```

create a file called api.ini in the directory where your python script will run to house the Looker API connection

parameters:

```
# Base URL for API. Do not include /api/* in the url
base_url = https://mylooker.looker.com:19999
# API 3 client id
client_id=put_your_client_id_here
# API 3 client secret
client_secret=put_your_sectret_here
# Set to false if testing locally against self-signed certs. Otherwise leave True
```

The automation python file follows these high level steps.

1. connect to the Looker API to pull a list of EAV fields
2. create a pyLookML project connection to your github
3. Set up the objects we'll be manipulating (some are just strings which will be added back to the LookML at the end)
4. loop over the list of EAV k,v pairs and do work
5. loop over the distinct raw columns (obtained in the full k,v loop) for adding columns to the NDT
6. loop over the distinct org ids to add the model's access grants
7. add all the final objects back to the model file
8. save the file back to the project in github
9. hit the looker deploy URL to sync Looker production mode with the github master branch

```
import lookml
from looker_sdk import models, methods, init40
import json

# step 1 -- connect to the Looker API to pull a list of EAV fields
sdk = init40("api.ini")
sql_for_fields = f"""
        SELECT
            cpf.org_id
            ,cpf.value
            ,cpf.datatype
            ,cpf.field_name as FIELD_NAME
            , CASE
                WHEN cpf.datatype IN ('TIMESTAMP_LTZ') THEN 'time'
                WHEN cpf.datatype IN ('FLOAT','NUMBER', 'int') THEN 'number'
                ELSE 'string' END as LOOKER_TYPE
        FROM
            -- public.custom_profile_fields as cpf
            (
                SELECT 1 as user_id, 8 as org_id, 'c_donation_amount' as field_name,
↪'40' as value, 'int' as datatype UNION ALL
                SELECT 1, 8, 'c_highest_achievement', 'gold badge', 'varchar' UNION␣
↪ALL
                SELECT 2, 101, 'c_highest_achievement', 'silver badge', 'varchar'␣
↪UNION ALL
                SELECT 2, 101, 'c_monthly_contribution', '300', 'int' UNION ALL
                SELECT 3, 101, 'c_highest_achievement', 'bronze badge', 'varchar'␣
↪UNION ALL
                SELECT 3, 101, 'c_monthly_contribution', '350', 'int' UNION ALL
                SELECT 4, 101, 'c_monthly_contribution', '350', 'int' UNION ALL
```

(continues on next page)

(continued from previous page)

```
                SELECT 4, 101, 'age', '32', 'int' UNION ALL
                SELECT 5, 102, 'c_monthly_contribution', '100', 'int'
            ) as cpf
        WHERE
            1=1
        GROUP BY 1,2,3,4,5
"""
query_config = models.SqlQueryCreate(sql=sql_for_fields, connection_id="snowlooker")
query = sdk.create_sql_query(query_config)
response = json.loads(sdk.run_sql_query(slug=query.slug, result_format="json"))

# step 2 -- create a pyLookML project connection to your github
proj = lookml.Project(
        #the github location of the repo
        repo= 'llooker/your_repo'
        #instructions on creating an access token: https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line
        ,access_token='your_access_token'
        #your Looker host
        ,looker_host="https://example.looker.com/"
        #The name of the project on your looker host
        ,looker_project_name="pylookml_testing_2"
        #You can deploy to branches other than master, a shared or personal branch if you would like to hop into looker, pull
        #remote changes and review before the code is committed to production
        ,branch='master'
)
#For simplicity of this example, all of the objects we're tracking will be contained in the model file, but for your needs can be split across the project.
modelFile = proj['eav_model.model.lkml']

# step 3 -- Set up the objects we'll be manipulating (some are just strings which will be added back to the LookML at the end)
#the EAV source view points to our custom_profile_fields database table
eavSource = modelFile['views']['eav_source']
#the user profile we'll call the "flattening NDT" since that's where our flattening logic lives
flatteningNDT = modelFile['views']['usr_profile']


#Ensure there is a hidden explore to expose the eav_souce transformations to our user_profile NDT
modelFile + f'''
    explore: _eav_flattener {{
        from: {eavSource.name}
        hidden: yes
    }}
'''
#Begin the derived table, will be added to as we loop through the fields
drivedtableString = f'''
    derived_table: {{
        explore_source: _eav_flattener {{
            column: user_id {{ field: _eav_flattener.user_id }}
            column: org_id {{ field: _eav_flattener.org_id }}
'''

#Set up a pair of list to track the unique org ids and column names
```

(continues on next page)

(continued from previous page)

```
#since the api query will be at a org / column level this allows us to "de-dupe"
orgIds, columns = [], []

# step 4 -- loop over the list of EAV k,v pairs and do work
for column in response:
    dimName = lookml.core.lookCase(column['FIELD_NAME'])
    orgIds.append(column['org_id'])
    columns.append(dimName)
    #Step 1) Add flattening measure to the EAV source table
    eavSource + f'''
            measure: {dimName} {{
                type: max
                sql: CASE WHEN ${{field_name}} = '{column['FIELD_NAME']}' THEN ${{value}} ELSE NULL END;;
            }}
    '''

    # Add to the NDT fields
    flatteningNDT + f'''
            dimension: {dimName}_org_{column['org_id']} {{
                label: "{dimName}"
                type: {column['LOOKER_TYPE']}
                sql: ${{TABLE}}.{dimName} ;;
                required_access_grants: [org_{column['org_id']}]
            }}
    '''
    if column['LOOKER_TYPE'] == "number":
        flatteningNDT + f'''
            measure: {dimName}_total_org_{column['org_id']} {{
                label: "{dimName}_total"
                type: sum
                sql: ${{{dimName}_org_{column['org_id']}}} ;;
                required_access_grants: [org_{column['org_id']}]
            }}
        '''
# step 5 -- loop over the distinct raw columns (obtained in the full k,v loop) for adding columns to the NDT
for col in set(columns):
    drivedtableString += f' column: {col} {{ field: _eav_flattener.{col} }}'
drivedtableString += '}}'

# step 6 -- loop over the distinct org ids to add the model's access grants
accessGrants = ''
for org in set(orgIds):
    accessGrants += f'''
        access_grant: org_{org} {{
        user_attribute: org_id
        allowed_values: [
            "{org}"
        ]
        }}
    '''
# step 7 -- add all the final objects back to the model file
#Finish by adding some of the strings we've been tracking:
flatteningNDT + drivedtableString
#Add access grants to the model
modelFile + accessGrants
```

(continues on next page)

(continued from previous page)

```
133
134  # step 8 -- save the file back to the project in github
135  proj.put(modelFile)
136  #s step 9 -- hit the looker deploy URL to sync Looker production mode with the␣
     ↪github master branch
137  proj.deploy()
```

The Completed LookML output to the eav.model.lkml file

```
connection: "snowlooker"

access_grant: org_8 {
    user_attribute: org_id
    allowed_values: [
        "8",
    ]
}
access_grant: org_101 {
    user_attribute: org_id
    allowed_values: [
        "101",
    ]
}
access_grant: org_102 {
    user_attribute: org_id
    allowed_values: [
        "102",
    ]
}

explore: usr {
    access_filter: {
        field: usr_profile.org_id
        user_attribute: org_id
    }
    join: usr_profile {
        type: left_outer
        relationship: one_to_one
        sql_on: ${usr.id} =  ${usr_profile.user_id} ;;
    }
}

explore: _eav_flattener {
    from: eav_source
    hidden: yes
}

view: usr {
    sql_table_name: public.users ;;
    dimension: email {}
    dimension: id {}
    dimension_group: created {
        timeframes: [
            raw, date, month, year,
        ]
        type: time
        }
```

(continues on next page)

(continued from previous page)

```
}

view: usr_profile {

derived_table: {
    explore_source: _eav_flattener {
    column: user_id { field: _eav_flattener.user_id}
    column: org_id { field: _eav_flattener.org_id }
    column: c_donation_amount { field: _eav_flattener.c_donation_amount}
    column: c_monthly_contribution { field: _eav_flattener.c_monthly_contribution }
    column: c_highest_achievement { field: _eav_flattener.c_highest_achievement }
    column: age { field: _eav_flattener.age }
    }
}
dimension: age_org_101 {
    label: "age"
    type: number
    sql: ${TABLE}.age ;;
    required_access_grants: [org_101,]
    }
dimension: c_donation_amount_org_8 {
    label: "c_donation_amount"
    type: number
    sql: ${TABLE}.c_donation_amount ;;
    required_access_grants: [org_8,]
    }
dimension: c_highest_achievement_org_101 {
    label: "c_highest_achievement"
    type: string
    sql: ${TABLE}.c_highest_achievement ;;
    required_access_grants: [org_101,]
    }
dimension: c_highest_achievement_org_8 {
    label: "c_highest_achievement"
    type: string
    sql: ${TABLE}.c_highest_achievement ;;
    required_access_grants: [org_8,]
    }
dimension: c_monthly_contribution_org_101 {
    label: "c_monthly_contribution"
    type: number
    sql: ${TABLE}.c_monthly_contribution ;;
    required_access_grants: [org_101,]
    }
dimension: c_monthly_contribution_org_102 {
    label: "c_monthly_contribution"
    type: number
    sql: ${TABLE}.c_monthly_contribution ;;
    required_access_grants: [org_102,]
    }
dimension: org_id {}
dimension: user_id {}
measure: age_total_org_101 {
    label: "age_total"
    type: sum
    sql: ${age_org_101} ;;
    required_access_grants: [org_101,]
```

(continues on next page)

(continued from previous page)

```
    }
measure: c_donation_amount_total_org_8 {
    label: "c_donation_amount_total"
    type: sum
    sql: ${c_donation_amount_org_8} ;;
    required_access_grants: [org_8,]
    }
measure: c_monthly_contribution_total_org_101 {
    label: "c_monthly_contribution_total"
    type: sum
    sql: ${c_monthly_contribution_org_101} ;;
    required_access_grants: [org_101,]
    }
measure: c_monthly_contribution_total_org_102 {
    label: "c_monthly_contribution_total"
    type: sum
    sql: ${c_monthly_contribution_org_102} ;;
    required_access_grants: [org_102,]
    }
}

view: eav_source {
sql_table_name: public.custom_profile_fields ;;
dimension: datatype { type: string }
dimension: field_name { type: string }
dimension: org_id { type: number }
dimension: user_id { type: number }
dimension: value { type: string }

measure: age {
    type: max
    sql: CASE WHEN ${field_name} = 'age' THEN ${value} ELSE NULL END ;;
    }
measure: c_donation_amount {
    type: max
    sql: CASE WHEN ${field_name} = 'c_donation_amount' THEN ${value} ELSE NULL END ;;
    }
measure: c_highest_achievement {
    type: max
    sql: CASE WHEN ${field_name} = 'c_highest_achievement' THEN ${value} ELSE NULL␣
↪END ;;
    }
measure: c_monthly_contribution {
    type: max
    sql: CASE WHEN ${field_name} = 'c_monthly_contribution' THEN ${value} ELSE NULL␣
↪END ;;
    }
}
```

More information and resources

1. 2019 Looker JOIN presentation on EAV and LookML Generation
2. More about modeling EAV data in Looker

As an alternative to the MAX(CASE WHEN NAME='foo' THEN VALUE END) construct, you can use first / last

value window functions. The specifics of the implementation may look slightly different.

```
FIRST_VALUE(
    CASE
        WHEN attributename = 'single_type' THEN attributevalue
        ELSE NULL
    END
IGNORE NULLS)
OVER (partition by sessionid order by sessionid)
```

3.4 Autotune your model using PyLookML

PyLookML offers a command line interface (CLI) which offers several commands, one of which is autotune. It will automatically create aggregate awareness tables inside of your LookML model based on the most frequently run queries and commit to a developer branch so that you can confirm the output first.

Let's get started with an example: Ensure that you have installed it using pip, which will bind the lookml command. **Note**: if you install it in a virtual environment the lookml command will only be available when the virtual environment is active.

```
pip install lookml
```

We will be using a cli command 'lookml autotune' which will search for a file in your current directory called autotune.ini.

- pyLookML look for an autotune.ini file in the current working directory
- Your autotune.ini should look like this:

```
[autotune]
access_token = xxx
looker_host = https://mycompany.looker.com:19999
api_client = xxx
api_secret = yyy
model_name = bike_share
branch = dev-john-doe-yddt
```

Then on the command line you can run:

```
lookml autotune
```

If your autotune.ini is stored in a different location, you can provide the path by running

```
lookml autotune useconfig
```

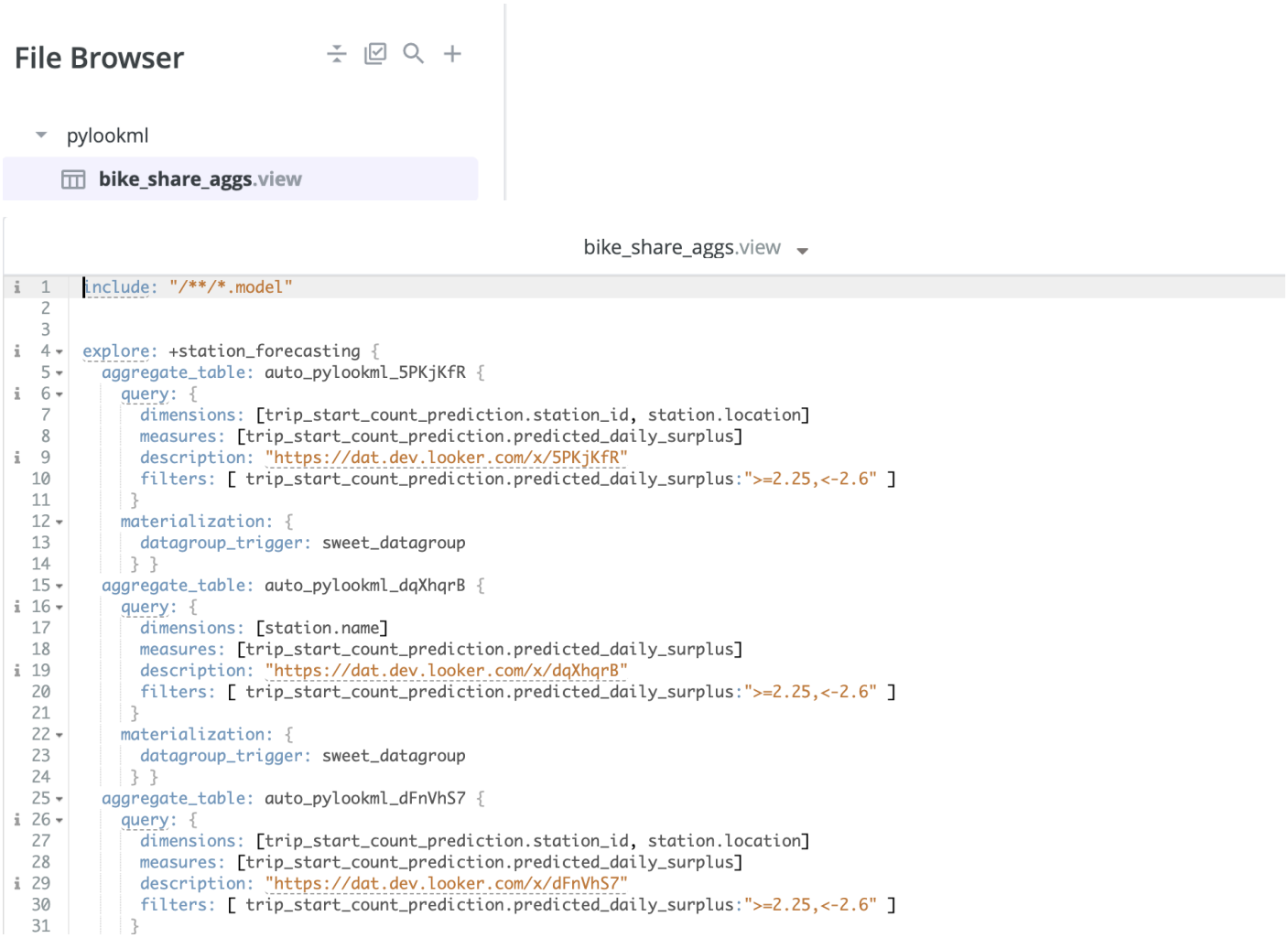
and you will be prompted to provide the path

If you would like to provide each bit of info interactively run:

```
lookml autotune guided
```

it will ask you for all the info and you can paste it in.

It may take a minute to run, but the result will be a single file with your aggregates located on the branch you provided, allowing you to check the output before pushing to production.



3.5 Full API Reference

3.6 Change Log

Starting with PyLookML version 3.0.0

3.0.3

- fixed an issue with the constructor not accepting lookML names with numbers Issue Link.

The following code now works:

```
my_dim = lookml.Dimension('dimension: custom_5 {}')
```

3.0.0

- complete and more stable re-write geared toward maximum backward compatibility
- language complete for all the latest LookML language updates (as of Looker 7.20) (new filters, materializations etc)
- significantly better whitespace handling

- can connect to filesystem without git
- added a CLI with various functions, including project dir list and autotune
- added new operator overloading syntax
- more helpful error messages
- options such as OMIT_DEFAULTS = true