heatmap-example

November 5, 2014

1 Overview

This notebook is a follow-up to the de-example.ipynb notebook.

The end goal is a heatmap of normalized H3K4me3 ChIP-seq signal in K562 cells, split into signal over genes that were up, down and unchanged compared to H1-hESC cells.

```
In [1]: # Show figures in the notebook
        %matplotlib inline
```

2 Imports

The imports are mostly the same as described in de-example.ipynb.

Here we use two additional imports:

- multiprocessing is a built-in module; here we will use it to figure out how many CPUs your computer has to take advantage of multiple cores
- datetime is another built-in module; here we will use it to display the current time to see how fast certain steps take to run.

```
In [2]: import metaseq
        import os
        from matplotlib import pyplot as plt
        import matplotlib
        import gffutils
        import pybedtools
        import numpy as np
        from pybedtools import featurefuncs as ff
        import multiprocessing
        import datetime

        # Set up an easier-to-type alias
        now = datetime.datetime.now

        # show what version of metaseq we're using
        print metaseq.__version__
```

```
0.5.4
```

3 Configuration

The following configuration is the same as in de-example.ipynb, that is, some tweaks to matplotlib's defaults and a performance tweak for sqlite3 databases on networked filesystems.

```
In [3]: plt.rcParams['font.family'] = 'Arial'
        plt.rcParams['font.size'] = 11
        plt.rcParams['legend.scatterpoints'] = 1
        plt.rcParams['legend.fontsize'] = 10
        cache_dir = '/tmp'
        dbfn = 'data/gencode.v19.annotation.chr11.db'
        deseq_results_filename = 'data/DESeq-results.txt'

        cached_db = os.path.join(cache_dir, os.path.basename(dbfn))
        if not os.path.exists(cached_db):
            os.system('cp -v %s %s' % (dbfn, cached_db))
```

3.1 Other settings

We will use the following settings for this demo. Placing them at the beginning here makes it easy to tweak and re-run later.

```
In [4]: # Distance up- and downstream from promoter to use as windows
        UPSTREAM = 5000
        DOWNSTREAM = 5000

        # Windows will be binned into this many bins
        BINS = 100

        # BAM file for the array
        BAM = 'data/K562_H3K4me3.chr11.bam'
        INPUT = 'data/K562_input.chr11.bam'

        # Number of CPUs to use to calculate array.  Here we get
        # the total number of CPUs on the machine
        CPUS = multiprocessing.cpu_count()
```

As before, we set up a `DESeq2Results` object pointing to the results, a `gffutils.FeatureDB` object pointing to the database, and attach the database.

```
In [5]: d = metaseq.results_table.DESeq2Results(deseq_results_filename)
        db = gffutils.FeatureDB(cached_db)
        d.attach_db(db)
```

4 Create genomic signal objects

`metaseq` is very flexible with what sorts of data to use as "genomic signal" by just providing a filename and a datatype. Current options are `'bam'`, `'bigwig'`, `'bigbed'`, `'bed''`, `'gtf'`, `gff'`, `bed'`, `'vcf'`.

We need to set up one `genomic_signal` object for the IP and one for the control (input).

```
In [6]: gs_ip = metaseq.genomic_signal(BAM, 'bam')
        gs_input = metaseq.genomic_signal(INPUT, 'bam')
```

5 Create "windows"

Now that we have defined what we want our signal to be, we need to make windows over which to view that signal.

We'll demonstrate two kinds of windows:

1) fixed windows, where we look at TSS +/- 5kb, and

2) variable-sized windows where we look each gene scaled from 0-100%.

In each case, we will generate a heatmap using 100 bins. So for the TSS heatmap, each row will represent one gene's TSS, and each column will represent 100 bp (+/-5kb split into 100 bins). For the gene heatmap, the columns will represent a different number of bp depending on how long each gene is.

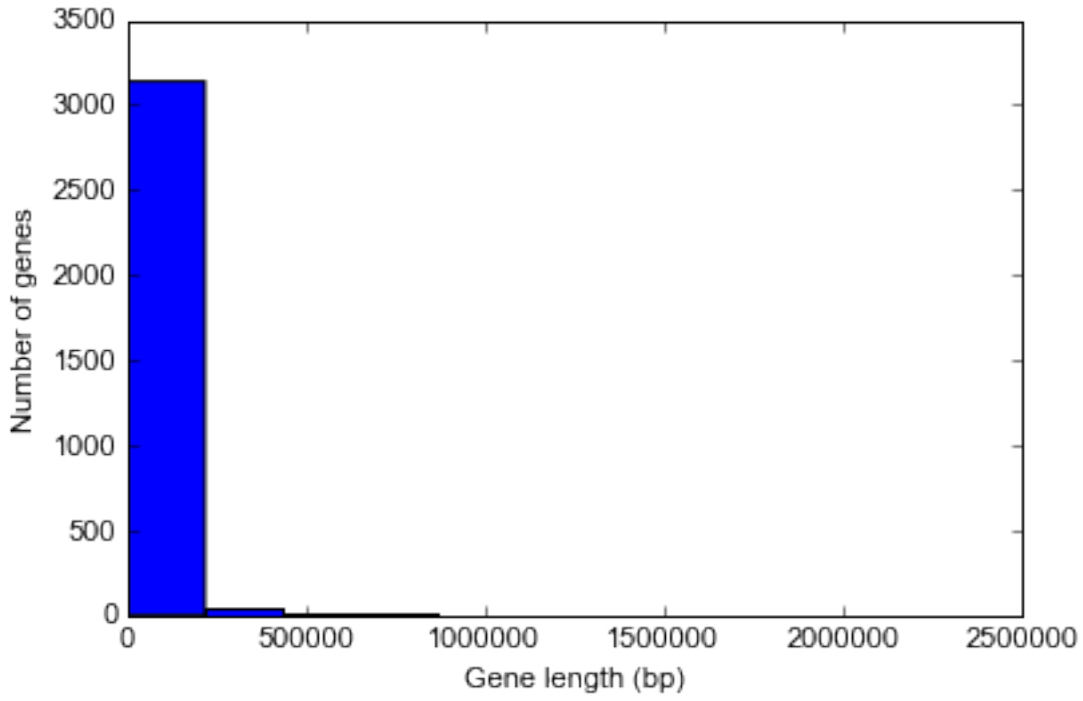
Just like in de-example.ipynb, we use `d.features()` to get the features in our differential expression data, and `pybedtools` to manipulate the intervals. Here, we save the results to filenames in case we want to use them again for downstream analysis.

```
In [7]: tss_filename = 'TSS.gtf'
        gene_filename = 'genes.gtf'

        genes = pybedtools.BedTool(d.features())\
            .saveas(gene_filename)

        tsses = genes\
            .each(ff.TSS, upstream=UPSTREAM, downstream=DOWNSTREAM)\
            .saveas(tss_filename)
```

6 Side note: QC on the genes

Here is a brief demo on using `pybedtools` for digging a little deeper into gene info. Recall that `genes` is a `pybedtools.BedTool` object, which can be iterated over to yield `pybedtools.Interval` objects. These, in turn, can be checked for their length. What does a histogram of gene lengths look like?

```
In [8]: gene_lengths = [len(i) for i in genes]
        plt.hist(gene_lengths);
        plt.xlabel('Gene length (bp)');
        plt.ylabel('Number of genes');
```



Wow, this histogram goes pretty high. What's the min/max?

```
In [9]: print "min length:" , min(gene_lengths)
        print "max length:", max(gene_lengths)
```

```
min length: 37
max length: 2172912
```

Hmm, what gene is 2.2 Mb? We can use `np.argmax` to get the index at which `gene_lengths` is highest, and then index into the `genes` object to get the interval. Printing the resulting `pybedtools.Interval` displays the feature as the equivalent line in the GTF file:

```
In [10]: longest_gene = genes[np.argmax(gene_lengths)]
         print longest_gene
```

```
chr11        HAVANA        gene        83166055        85338966        .        -        .        gene_s
```

Perhaps we want to do some filtering to remove very long or very short genes, which might cause some artifacts in the heatmap. Here's an easy way to do that with `pybedtools`. Note that it's easier to make that filter function more sophisticated – like only the protein coding genes of a certain length (by adding `and f['gene_type'] == "protein_coding"`). For now, let's get rid of genes that are over 100kb.

```
In [11]: def length_filter(f):
             if len(f) < 100000:
                 return True

         filtered_genes = genes.filter(length_filter).saveas('filtered-genes.gtf')
         print "%s of %s genes filtered out" % (len(genes) - len(filtered_genes), len(genes))
```

```
193 of 3208 genes filtered out
```

Beware! Now the genes in `filtered_genes` no longer match the genes in the `DESeq2Results` object, `d`.

```
In [12]: print "len(d):", len(d)
         print "len(filtered_genes):", len(filtered_genes)
```

```
len(d): 3208
len(filtered_genes): 3015
```

The `reindex_to()` method lets us re-align the `DESeq2Results` object to any arbitrary GTF or GFF file, as long as the features in that file have an attribute that contains the gene ID. In `filtered_genes`, that attribute is `"gene_id"` so we can do this:

```
In [13]: filtered_d = d.reindex_to(filtered_genes, attribute='gene_id')
         print "old:", len(d)
         print "new:", len(filtered_d)
```

```
old: 3208
new: 3015
```

This shows how to filter features by arbitrarily complex criteria and then re-align the expression data with the genomic interval data to ensure it matches.

However, for simplicity we will continue to use the original unfiltered genes along with the unfiltered `DESeq2Results` object.

7 Create arrays

OK, we have genomic signal, and we have windows over which to view the signal. Let's make some arrays that we can visualize as heatmaps.

Since we're doing two different versions of the heatmap – TSS and full gene – it makes sense to group together the common code into a single function (here, called `make_arrays`).

Note that to speed up future runs, we store the results to disk. If the filename for the stored results already exists, this step will be skipped. While this saves minimal time for this example containing only data for chr11, it can save a lot of time when working with many full-genome data sets.

Also note that this function normalizes the IP and control by million mapped reads. At the end, normalized input is subtracted from the normalized IP, to give a final normalized array of enrichment.

```
In [14]: def make_arrays(prefix, features):
             """
             If `prefix.npz` doesn't exist, then create arrays
             using `features` and save to `prefix.npz`.

             The H3K4me3 and input arrays are normalized by million
             mapped reads and stored in the .npz file under the keys
             'h3k4me3' and 'control', respectively.

             After creation (or if the file already exists), the IP
             and control arrays are loaded, and control is subtracted
             from IP.  The features and this normalized array are
             then returned.
             """
             fragment_size = 200

             # If the filename already exists, then don't do anything.
             if os.path.exists(prefix + '.npz'):
                 print now(), "%s.npz already exists; using that" % prefix

             # Otherwise we have some work to do...
             else:
                 # ensure that ip and control are always created with the
                 # same keyword arguments
                 kwargs = dict(features=features, bins=BINS,
                               processes=CPUS, fragment_size=fragment_size)

                 # create the IP array for whatever features were provided
                 print now(),
                 print "building the IP array for %s using %s processors" % (prefix, CPUS)
                 a = gs_ip.array(**kwargs)

                 # normalize by million mapped reads
                 a = a / gs_ip.mapped_read_count() * 1e6

                 # Same thing for the control data
                 print now(),
                 print "building the input array for %s using %s processors" % (prefix, CPUS)
                 i = gs_input.array(**kwargs)
                 i = i / gs_input.mapped_read_count() * 1e6

                 # here's where we decide what to label the arrays in the .npz file
                 arrays = dict(h3k4me3=a, control=i)
```

```
            # save arrays and features to disk
            metaseq.persistence.save_features_and_arrays(
                features=features,
                arrays=arrays,
                prefix=prefix)
            print now(), "done"


        # Now load the data from disk
        loaded_features, loaded_arrays = \
            metaseq.persistence.load_features_and_arrays(prefix=prefix)

        # Subtract control from ip to get the final normalized array
        normalized_array = loaded_arrays['h3k4me3'] - loaded_arrays['control']

        return loaded_features, normalized_array


    # Now we call our new function for the TSSs:
    tss_features, tss_array = make_arrays(
        prefix='tss_%s_%s' % (UPSTREAM, DOWNSTREAM),
        features=tsses)

    # And genes:
    gene_features, gene_array = make_arrays(
        prefix='gene', features=genes)
```

```
2014-11-05 13:27:20.751800 building the IP array for tss_5000_5000 using 8 processors
2014-11-05 13:27:25.274100 building the input array for tss_5000_5000 using 8 processors
2014-11-05 13:27:27.400596 done
2014-11-05 13:27:27.419652 building the IP array for gene using 8 processors
2014-11-05 13:27:31.607938 building the input array for gene using 8 processors
2014-11-05 13:27:35.123687 done
```

The array creation is highly dependent on the number of CPUs available. Using 6+ CPUs, it takes about 1 second per 1000 features.

8 Plotting

Next, we need to create the x-values for our arrays. The TSSes go upstream and downstream by 5kb, while the gene array goes from 0 to 100%.

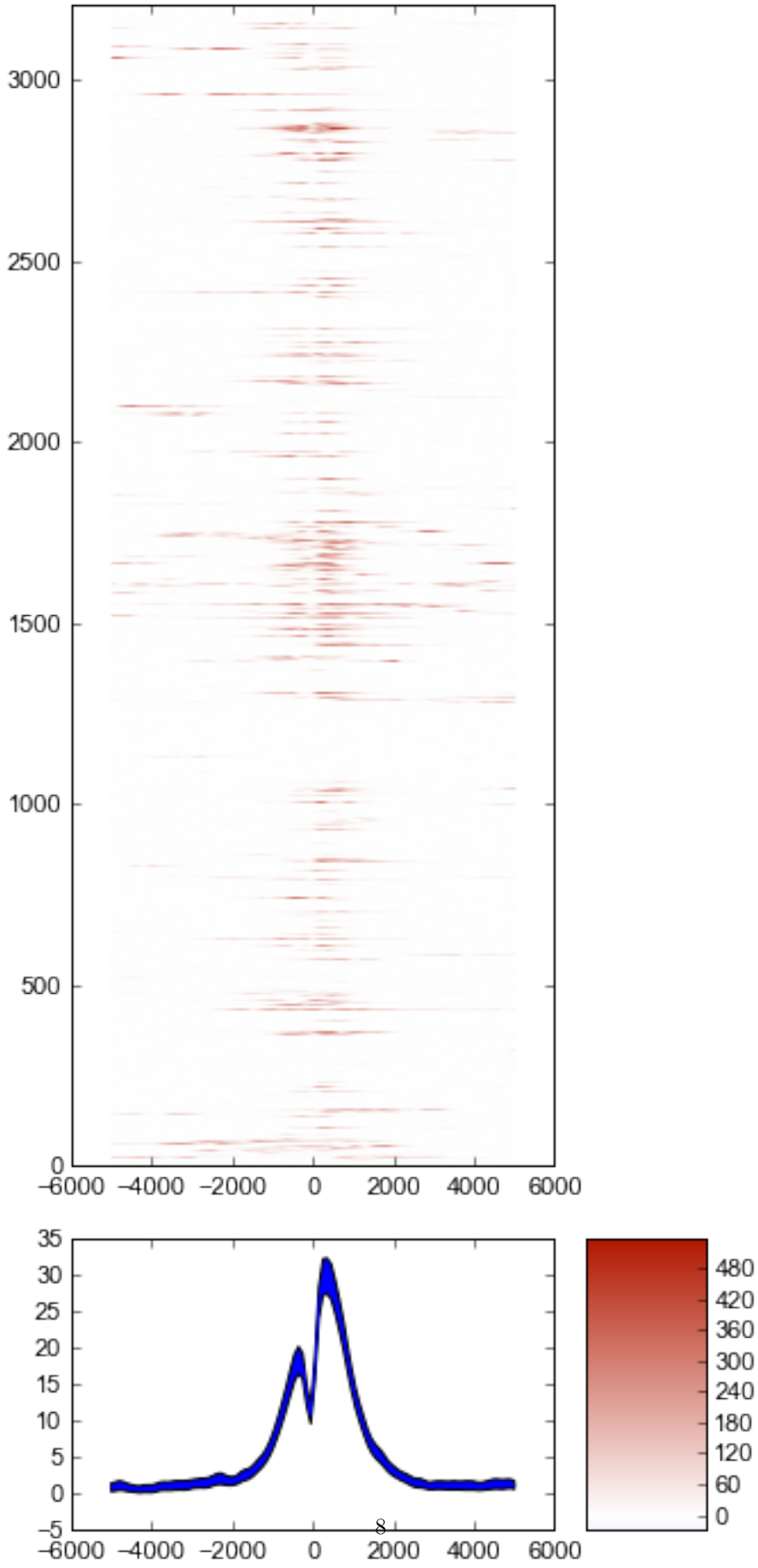
```
In [15]: tss_x = np.linspace(-UPSTREAM, DOWNSTREAM, BINS)
         gene_x = np.linspace(0, 100, BINS)
```

Let's work with just the TSS version for now in order to demonstrate some of the plotting tools. We will be using the `metaseq.plotutils.imshow` function, which sets up a lot of things behind the scenes. For example:

- a figure is set up with 3 axes: the heatmap, a colorbar, and axes on which the average signal is plotted as a line plus calculated 95% confidence intervals.
- a colormap is created going from blue to red, with white centered on zero
- the heatmap and line plot axes are connected, such that zooming in on one will zoom in on the other

In its simplest form, just give it the array and the x values:

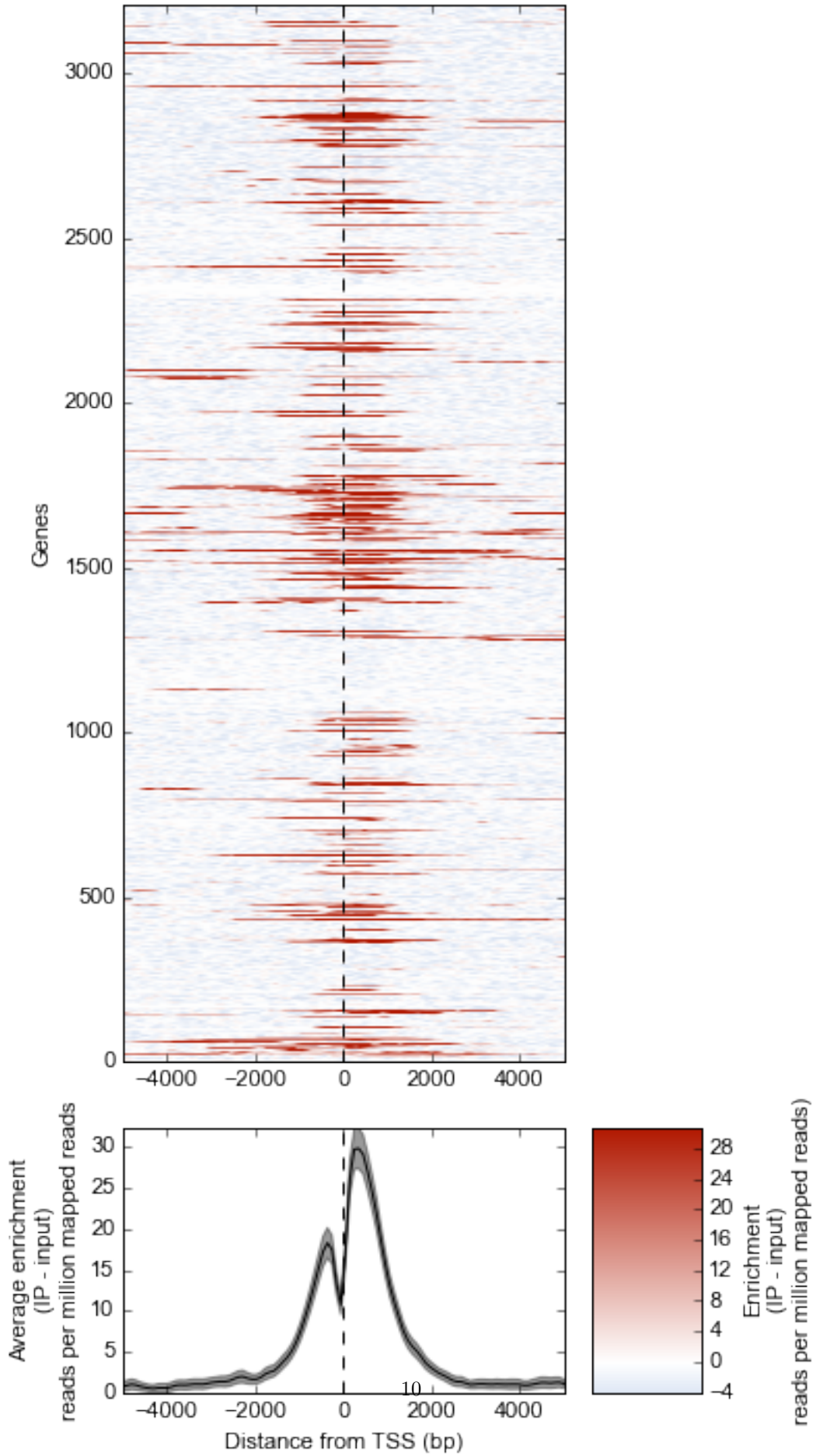
```
In [16]: # First pass
         fig = metaseq.plotutils.imshow(
             tss_array,
             x=tss_x,
         )
```


3000
2500
2000
1500
1000
500
0
−6000 −4000 −2000 0 2000 4000 6000
35
30
25
20
15
10
5
0
−5
−6000 −4000 −2000 0 2000 4000 6000
480
420
360
300
240
180
120
60
0

This could use a lot of tweaking; luckily `metaseq` is built to allow a lot of tweaking . . .
In this next code block, we:

- increase the contrast by changing the limits of the colorbar to the 5th/95th percentiles (instead of the default min/max).
- change the color of the line plot using the `line_kwargs` and `fill_kwargs` arguments to `metaseq.plotutils.imshow`.
- once the figure is created we draw vertical lines indicating the TSS.
- finally, we properly label all the axes. Note that the `Figure` object returned (here we store it as `fig`) has the attributes `fig.array_axes`, `fig.line_axes`, and `fig.cax` which correspond to each respective `matplotlib.Axes` object, enabling lots of customization.

```
In [17]: fig = metaseq.plotutils.imshow(
             tss_array,
             x=tss_x,

             # Increase the contrast by truncating the colormap
             # to 5th and 95th percentiles
             vmin=5,
             vmax=95,
             percentile=True,

             # tweak the line plot
             line_kwargs=dict(color='k'),
             fill_kwargs=dict(color='k', alpha=0.4)
         )

         # auto-zoom axes
         fig.line_axes.axis('tight')

         # draw a vertical line at zero on both axes
         for ax in [fig.line_axes, fig.array_axes]:
             ax.axvline(0, color='k', linestyle='--')

         # Label axes
         fig.array_axes.set_ylabel("Genes")
         fig.line_axes.set_ylabel("Average enrichment\n(IP - input)\nreads per million mapped reads")
         fig.line_axes.set_xlabel("Distance from TSS (bp)")
         fig.cax.set_ylabel("Enrichment\n(IP - input)\nreads per million mapped reads)");
```


Genes
3000
2500
2000
1500
1000
500
0
−4000
−2000
0
2000
4000
Average enrichment
(IP - input)
reads per million mapped reads
30
25
20
15
10
5
0
Distance from TSS (bp)
28
24
20
16
12
8
4
0
−4
Enrichment
(IP - input)
reads per million mapped reads)

Looking better. As we might expect, H3K4me3 is stronger downstream of the promoter than upstream, and there's a distinct gap right at the promoter. Pan-H3 might be a better control than input, since this gap might be simply from a lack of H3-containing nucleosomes.

9 Tying in the expression data

Anyway, let's connect these data to the expression data. Let's make 3 clusters based on whether a gene went up, down, or was unchanged.

To do this, we have to label each row in the array according to its expression status:

```
In [18]: # everything's zero to start
         cls = np.zeros(len(d)).astype('str')
         PADJ = 0.05

         # up, down, unchanged
         subset = (
             ('unchanged', d.unchanged(PADJ).values),
             ('down', d.downregulated(PADJ).values),
             ('up', d.upregulated(PADJ).values),
         )
         for label, ind in subset:
             cls[ind] = label

         # make sure we didn't miss anything
         assert sum(cls == '0.0') == 0
```

So now `cls` is a vector of values corresponding to whether each gene was up, down, or unchanged. Here's the first 25 values:

```
In [19]: cls[:25]
```

```
Out[19]: array(['unchanged', 'unchanged', 'unchanged', 'unchanged', 'unchanged',
                'unchanged', 'unchanged', 'up', 'unchanged', 'unchanged',
                'unchanged', 'unchanged', 'up', 'unchanged', 'up', 'unchanged',
                'up', 'unchanged', 'unchanged', 'unchanged', 'unchanged',
                'unchanged', 'up', 'up', 'unchanged'],
               dtype='|S32')
```

10 Recap on interval-based data and ID-based data

Let's trace our steps backwards. We now have `cls`, which tells us the differential expression status of each gene. We got the differential expression status from `d`, the `DESeq2Results` object.

Also recall that when we created the array in the first place, we needed a set of features to use as windows. Those features came from calling `d.features()`. This means that each gene in the `d` object corresponds to the same row in the array, and that there is a 1:1 mapping between them.

When you do more complex analyses, keep in mind that you can use the `d.reindex_to` method to re-align the `DESeq2Results` object to the features you used to create the array.

11 Sorting and subsetting the heatmap

Let's create a score to use for sorting the heatmap. Here, we use the row mean. We could use things like the row max, or the median, or even expression values from the `DESeq2Results` object or the length of each gene, and these alternative scores are shown below as well if you'd like to experiment with them:

```
In [20]: # construct a score
         mean_score = tss_array.mean(axis=1)

         # examples of alternative scores:
         max_score = np.max(tss_array, axis=1)
         median_score = np.median(tss_array, axis=1)
         downstream_1kb_weighted_score = tss_array[:, 50:60].mean(axis=1)
         expression_score = d.data.baseMean
         change_score = d.data.padj
         length_score = [len(i) for i in gene_features]
         random_score = np.random.rand(len(tss_array))
         tip_zscores = metaseq.plotutils.tip_zscores(tss_array)
```

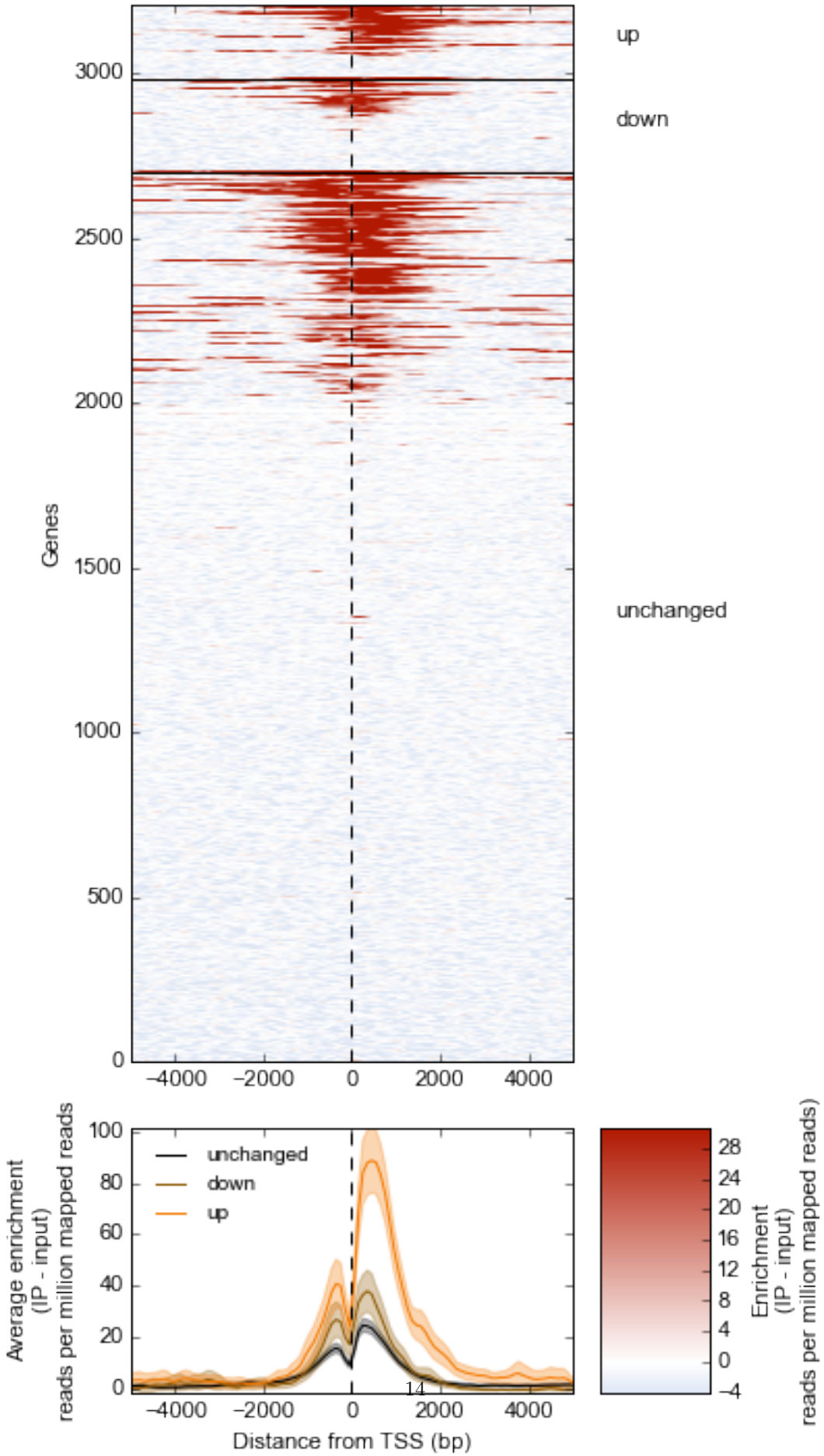
Finally, we define the order in which we'd like our subsets to appear. Since zero is on the bottom of the heatmaps by default, this subset order defines the order from bottom to top.

```
In [21]: subset_order = ['unchanged', 'down', 'up']
```

Now we keep everything the same as before, but this time including the new arguments `sort_by`, `subset_by`, and `subset_order`. Here, we also provide custom styles to the lines in the line axes, one style for each item in `subset_by`.

```
In [22]: up_color = '#f57900'
         dn_color = '#8f5902'
         un_color = '#000000'

         fig = metaseq.plotutils.imshow(
             # --------------------------------
             # This is the same as before...
             tss_array,
             x=tss_x,
             vmin=5,
             vmax=95,
             percentile=True,
             # --------------------------------


             # New stuff:
             #
             # Before, we only provided one style.  Here, we can provide
             # one style for each subset group.
             line_kwargs=[
                 dict(color=un_color, label='unchanged'),
                 dict(color=dn_color, label='down'),
                 dict(color=up_color, label='up'),
             ],

             fill_kwargs=[
                 dict(color=un_color, alpha=0.3),
                 dict(color=dn_color, alpha=0.3),
                 dict(color=up_color, alpha=0.3),
             ],

             # Sort the array by the score...
             sort_by=mean_score,
```

```
    # ...then subset by class...
    subset_by=cls,

    # ...and show the subsets in this order.
    subset_order=subset_order,
)

# ------------------------------------
# This is all the same as before...
fig.line_axes.axis('tight')
for ax in [fig.line_axes, fig.array_axes]:
    ax.axvline(0, color='k', linestyle='--')
fig.array_axes.set_ylabel("Genes")
fig.line_axes.set_ylabel("Average enrichment\n(IP - input)\nreads per million mapped reads")
fig.line_axes.set_xlabel("Distance from TSS (bp)")
fig.cax.set_ylabel("Enrichment\n(IP - input)\nreads per million mapped reads)")
# ------------------------------------

# New stuff:
#
# We use a helper function to add labels along the right-hand side
# of the array axes
metaseq.plotutils.add_labels_to_subsets(
    fig.array_axes,
    subset_by=cls,
    subset_order=subset_order,
)

# And add a legend to the line axes.  The lines are labeled according
# to what we used as the "label" key in each dictionary of
# "line_kwargs" above.
fig.line_axes.legend(loc='best', frameon=False);
```


up
down
unchanged
Genes
3000
2500
2000
1500
1000
500
0
−4000
−2000
0
2000
4000
unchanged
down
up
Average enrichment
(IP - input)
reads per million mapped reads
100
80
60
40
20
0
Distance from TSS (bp)
Enrichment
(IP - input)
reads per million mapped reads)
28
24
20
16
12
8
4
0
−4

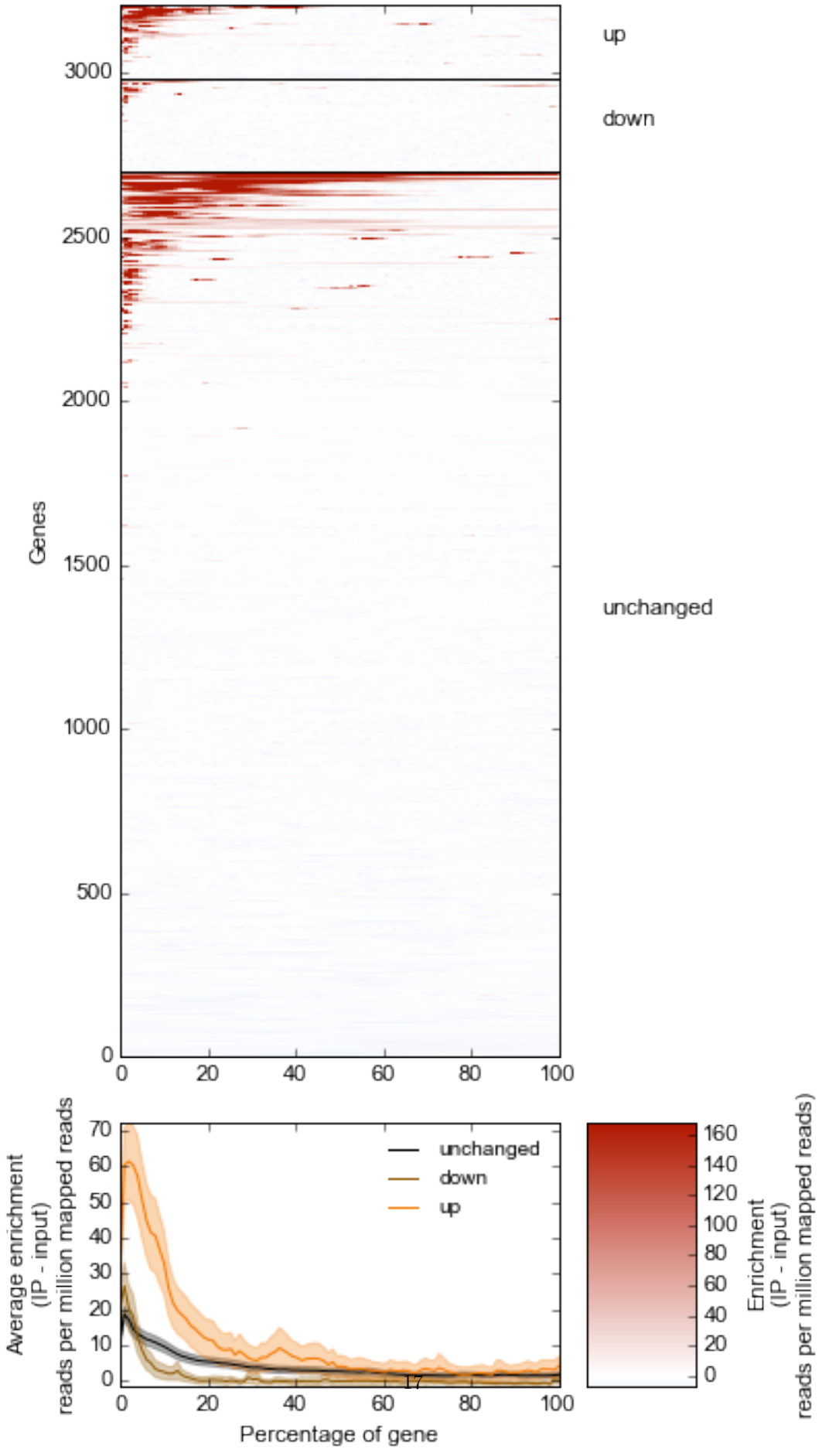
In all cases, we seem to be getting H3K4me3 signal only about 2kb into the gene body.

12 Same thing, but for the gene array

Now that we're happy with the plot, we can simply do the same thing for the gene array. We only need to change 3 things: 1) the array that is given to `metaseq.plotutils.imshow` (here we give it `gene_array` instead of `tss_array`), 3) the x values (0-100% instead of +/-5kb) and 2) the x-axis label.

In [23]:
```
# construct a score for the gene array
mean_score = gene_array.mean(axis=1)

fig = metaseq.plotutils.imshow(

    # This is the only thing that needs to change!
    gene_array,
    x=gene_x,

    # ------------------------------------------------
    # Everything else the same as for the TSS plot...

    vmin=1,  # less contrast for the gene plots
    vmax=99,
    percentile=True,
    line_kwargs=[
        dict(color=un_color, label='unchanged'),
        dict(color=dn_color, label='down'),
        dict(color=up_color, label='up'),
    ],
    fill_kwargs=[
        dict(color=un_color, alpha=0.3),
        dict(color=dn_color, alpha=0.3),
        dict(color=up_color, alpha=0.3),
    ],
    sort_by=mean_score,
    subset_by=cls,
    subset_order=subset_order,
    imshow_kwargs=dict(interpolation='none')
)

fig.line_axes.axis('tight')
for ax in [fig.line_axes, fig.array_axes]:
    ax.axvline(0, color='k', linestyle='--')
fig.array_axes.set_ylabel("Genes")
fig.line_axes.set_ylabel("Average enrichment\n(IP - input)\nreads per million mapped reads")
fig.line_axes.set_xlabel("Distance from TSS (bp)")
fig.line_axes.legend(loc='best', frameon=False)
fig.cax.set_ylabel("Enrichment\n(IP - input)\nreads per million mapped reads)")
metaseq.plotutils.add_labels_to_subsets(
    fig.array_axes,
    subset_by=cls,
    subset_order=subset_order,
)
```

```
# And here we reset the x axes label to be correct for the gene
# array we're using
fig.line_axes.set_xlabel("Percentage of gene");
```

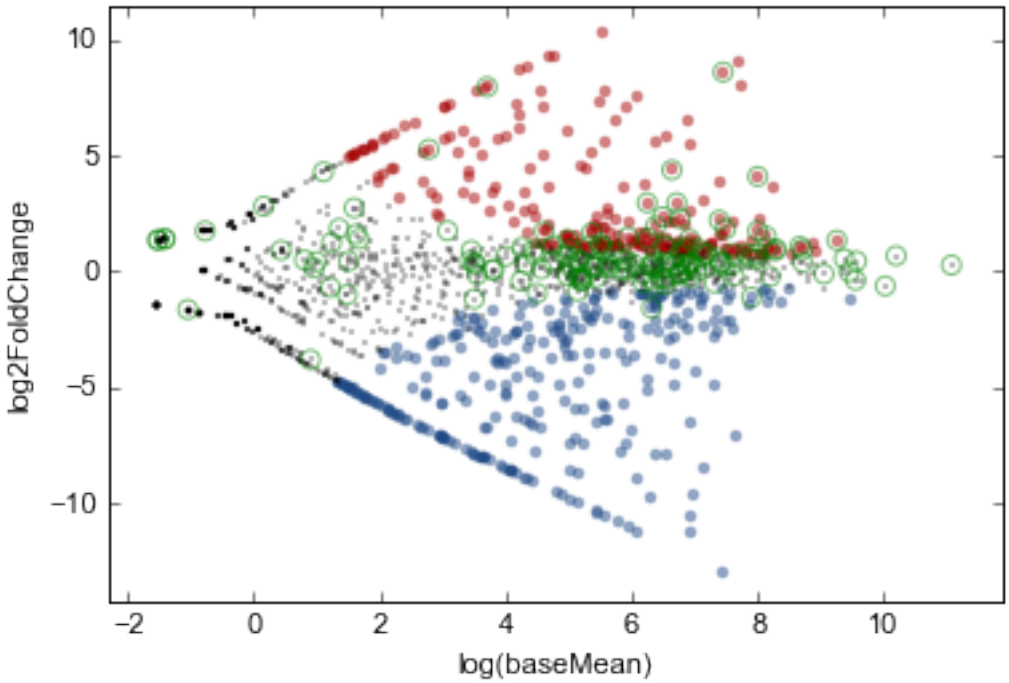

up
down
unchanged
Genes
3000
2500
2000
1500
1000
500
0
0
20
40
60
80
100
unchanged
down
up
Average enrichment
(IP - input)
reads per million mapped reads
70
60
50
40
30
20
10
0
Percentage of gene
160
140
120
100
80
60
40
20
0
Enrichment
(IP - input)
reads per million mapped reads)

Like the TSS-centric heatmap, when we stretch genes from 0-100% we also see a higher signal in the upregulated genes.

13 Connecting ChIP-seq data to expression data

In the de-example.ipynb example, we highlighted genes in the scatterplot that had called peaks nearby. To demonstrate how to integrate heatmap data with the expression data, let's highlight genes in the scatterplot whose mean H3K4me3 signal in both gene body and at the TSS is in the top 10% of all genes:

In [24]:
```
tss_top_decile_h3k4me3 = tss_array.mean(axis=1) > (np.percentile(tss_array.mean(axis=1), 90))
gene_top_decile_h3k4me3 = gene_array.mean(axis=1) > (np.percentile(gene_array.mean(axis=1), 90)

top_decile_h3k4me3 = tss_top_decile_h3k4me3 & gene_top_decile_h3k4me3

# This is mostly the same as in de-example.ipynb...
ax = d.scatter(
    xfunc=np.log,
    yfunc=None,
    genes_to_highlight=[
        (
            d.upregulated(.05),
            dict(color='#a40000', alpha=0.5)
        ),
        (
            d.downregulated(0.05),
            dict(color='#204a87', alpha=0.5)
        ),

        (
            d.unchanged(0.05),
            dict(color='k', alpha=0.3, marker='.')
        ),

        # ..except for this bit highlighting the genes with high H3K4me3
        (
            top_decile_h3k4me3,
            dict(color='g', s=50, facecolors='None', linewidth=1, alpha=0.5),
        ),


    ],
    x='baseMean',
    y='log2FoldChange',
    marginal_histograms=False,
);
```


10
5
0
−5
−10
log2FoldChange
−2
0
2
4
6
8
10
log(baseMean)