Electronic Supplementary Material (ESI) for Molecular Systems Design & Engineering.

Electronic Supporting Information

This jupyter notebook was created with python 3.6. In order to succesfully execute the cells you will need to install RDKit (http://www.rdkit.org/docs/Install.html) and PyChem (https://code.google.com/archive/p/pychem/downloads).

Table of Contents

Additional files attached to ESI of this manuscript:

- ESI.ipynb - this is the executable version of this document and can be run with:
  - `ipython notebook` (python 2.x) or
  - `jupyter notebook` (python 3.x)
- ESI.py - this python script contains in-house code to translate IUPAC names to SMILES representations for the salts featured in this work.
- saltInfo.csv - this csv file contains the IUPAC/SMILES data required to run ESI.py
- viscosity_processed.csv - this csv file contains the processed viscosity data with scaled and centered descriptors. A full description of the file is in the main manuscript
- viscosity_data.csv - this csv file contains the raw viscosity data: cation, anion, category, temperature, pressure, viscosity, and reference. A full description of the file is in the main manuscript

back to top

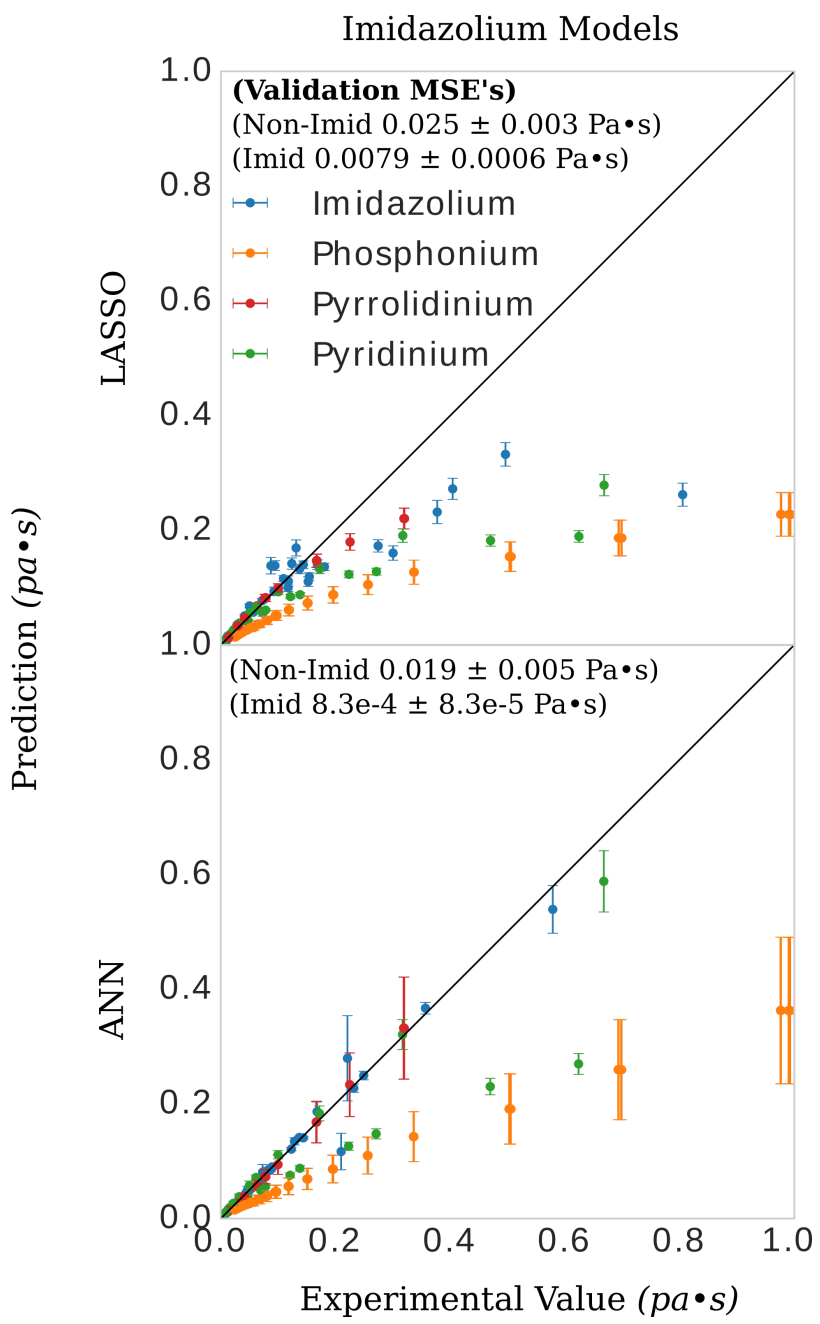


Figure S1. The predictions and standard deviations are obtained in the same way as discussed in Fig. 5 in the main manuscript. Notably, the standard deviations for a given salt-type increase with increasing viscosity—an artifact of the absolute value of those viscosities i.e. the percent variance is about the same. We also see that the ANN model is fairly accurate for some salts not included in its training data: the pyrrolidinium salts and some of the pyridinium salts. Both models produce poor predictions for phosphonium salt-types, especially in the high viscosity regions where data is more scarce.

back to top

Table S1. Error and standard deviation of validation set predictions from bootstrap for the all-salts ANN model (from bottom right panel of Fig. 5 in the manuscript)

RAAD	Temperature (K)	IL type
7.1 +/- 1.3	All	All
9.4 +/- 1.9	All	Imidazolium
1.8 +/- 0.1	All	Phosphonium
2.2 +/- 0.7	All	Pyrrolidinium
4.9 +/- 2.3	All	Pyridinium
14.9 +/- 4.8	273-298	All
5.8 +/- 1.8	299-323	All
3.7 +/- 0.7	324-348	All
4.6 +/- 2.2	349-373	All
21.5 +/- 7.6	273-298	Imidazolium
7.3 +/- 2.6	299-323	Imidazolium
4.5 +/- 1.0	324-348	Imidazolium
6.3 +/- 3.4	349-373	Imidazolium
6.3 +/- 3.4	273-298	Phosphonium
1.6 +/- 0.2	299-323	Phosphonium
2.2 +/- 0.1	324-348	Phosphonium
1.7 +/- 1.2	349-373	Phosphonium
2.5 +/- 0.8	273-298	Pyrrolidinium
1.8 +/- 0.7	299-323	Pyrrolidinium
N/A	324-348	Pyrrolidinium
N/A	349-373	Pyrrolidinium
11.9 +/- 6.9	273-298	Pyridinium
3.5 +/- 2.0	299-323	Pyridinium
3.6 +/- 3.4	324-348	Pyridinium
N/A	349-373	Pyridinium

Descriptors

Here we explain in detail the equations/methods employed in the selected descriptors for our viscosity models.

In [1]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from ESI import checkName

%matplotlib inline

anion_smiles = set(['CS(=O)(=O)O',
                    'C(F)(F)(F)S(=O)(=O)[O-]',
                    'CCOS(=O)(=O)[O-]',
                    'C(=C(C#N)C#N)=[N-]',
                    '[B-](F)(F)(F)F',
                    'C(C(F)(F)[P-](C(C(F)(F)F)(F)F)(C(C(F)(F)F)(F)F)(F)(F)F)(F)(F)F',
                    'C(#N)[S-]',
                    'C(#N)[N-]C#N',
                    'OCl(=O)(=O)=O',
                    'CC(=O)[O-]',
                    'C(C(F)(F)S(=O)(=O)NS(=O)(=O)C(C(F)(F)F)(F)F)(F)(F)F',
                    'F[P-](F)(F)(F)(F)F', 'C(F)(F)(F)S(=O)(=O)[N-]S(=O)(=O)C(F)(F)F',
                    'COS(=O)(=O)[O-]',
                    'C(C(F)(F)S(=O)(=O)[N-]S(=O)(=O)C(C(F)(F)F)(F)F)(F)(F)F',
                    'C(=O)(C(F)(F)F)[O-]'])
cation_smiles = set(['CCCC[N+]1=CC=CC=C1',
                     'CCCCN1C=C[N+](=C1C)C',
                     'CCCN1C=C[N+](=C1C)C',
                     'CCCC[N+]1=CC=CC(=C1)C',
                     'CCCC[N+]1(CCCC1)C',
                     'CCN1C=C[N+](=C1)C',
                     'CN1C=C[N+](=C1)C',
                     'CCCCCCCC[N+]1=CC=CC(=C1)C',
                     'CCCC[N+]1=CC=C(C=C1)C',
                     'CCCCN1C=C[N+](=C1)C',
                     'CCCCCCN1C=C[N+](=C1)C',
                     'CCCCCCCCCCCCCC[P+](CCCCCC)(CCCCCC)CCCCCC',
                     'CCCCCCCCN1C=C[N+](=C1)C'])
```

Nhyd

back to top

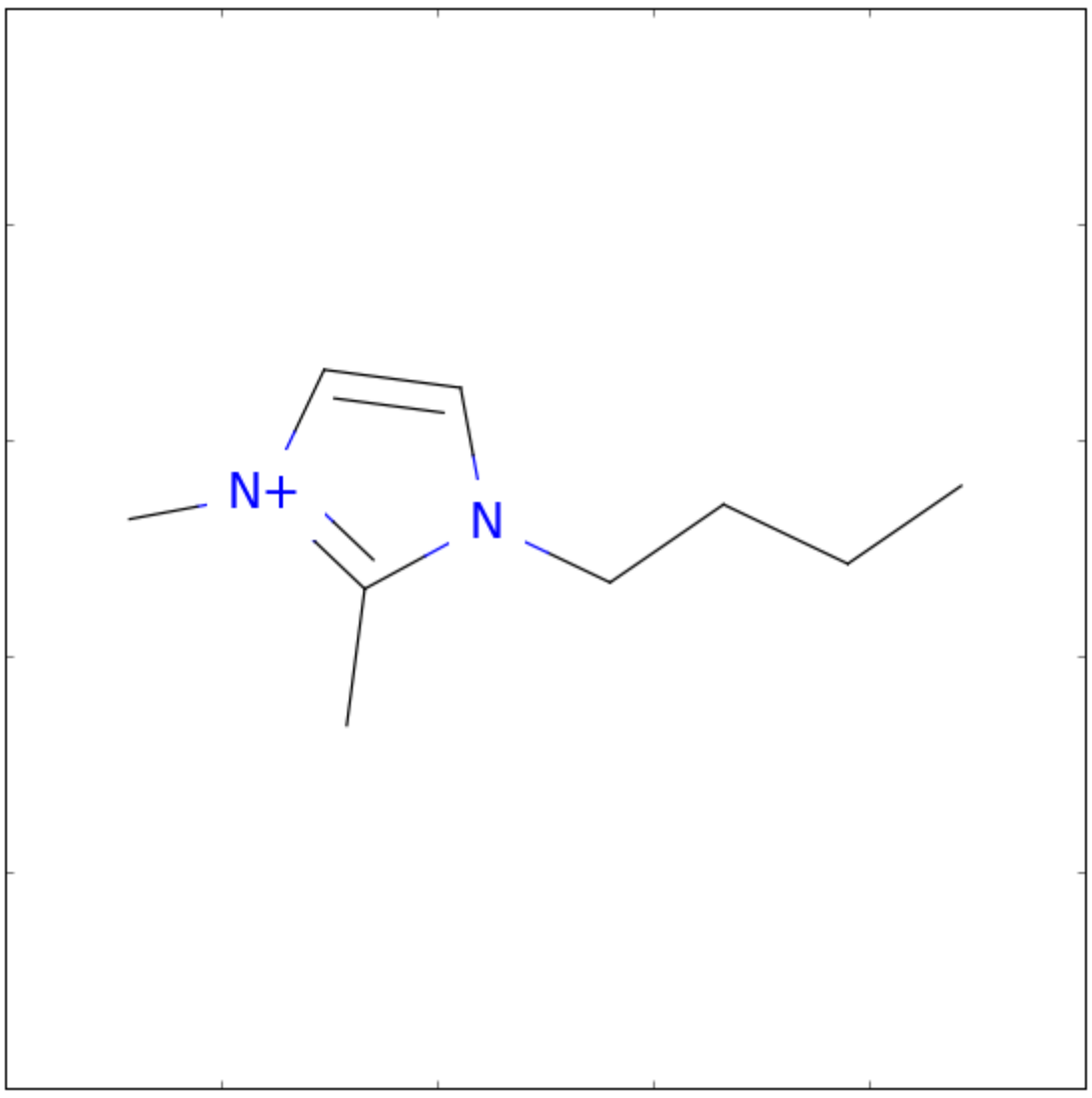
Class: constitutional

Nhyd is symply the sum of hydrogens in the molecule:

$$\sum H_{atom}$$

In [2]:
```
from rdkit import Chem
from rdkit.Chem import Draw
m = checkName("1-butyl-2,3-dimethyl-1H-imidazolium ")
mol = Chem.MolFromSmiles(m)
a = Draw.MolToMPL(mol)
Hmol = Chem.AddHs(mol)
i=0
for atom in Hmol.GetAtoms():
    if atom.GetSymbol() =='H':
        i+=1
print("{}\t{}".format("Number of Hydrogens:",i))
```

```
Number of Hydrogens:    17
```



Hato

back to top

Class: topological

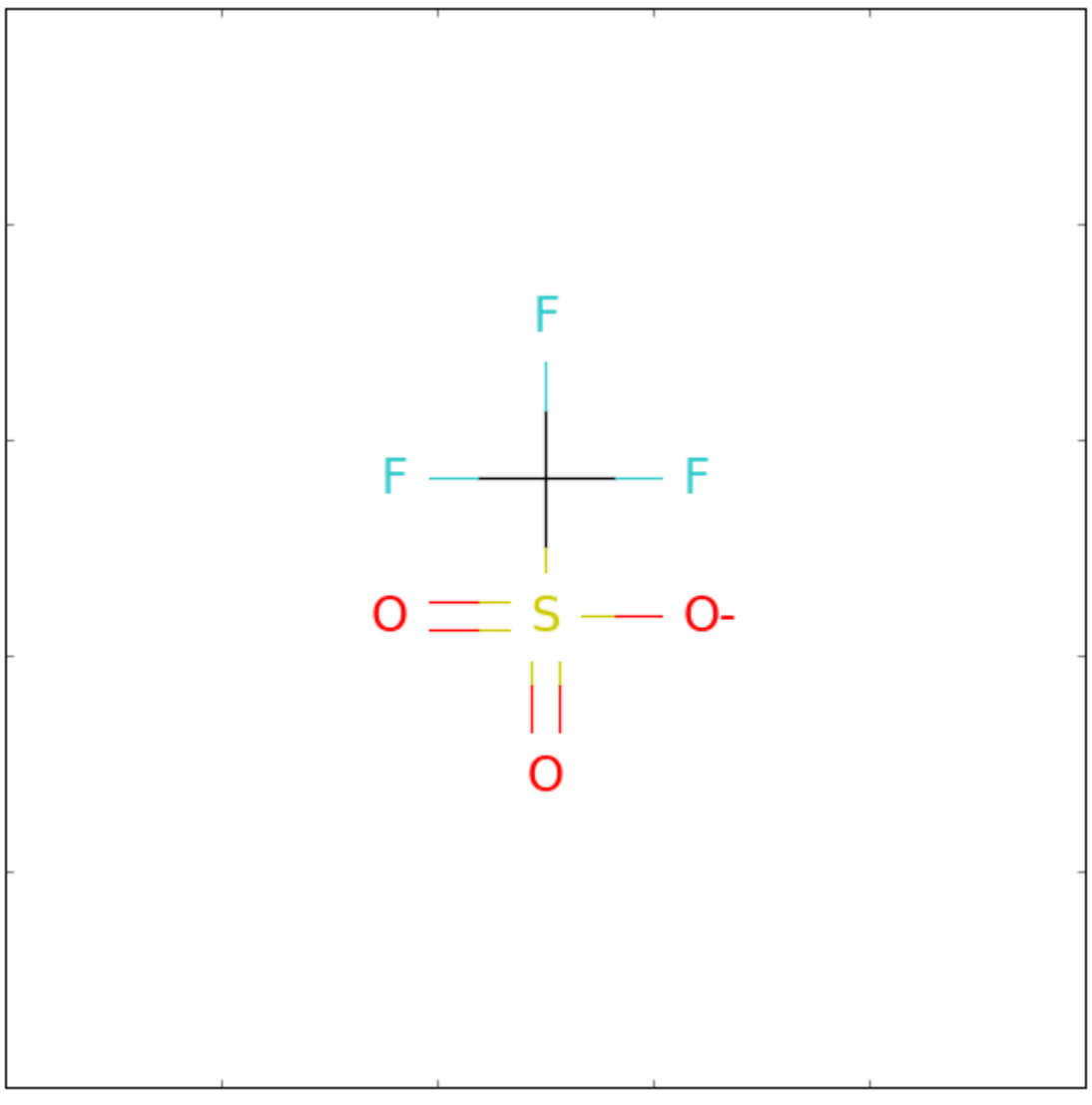
Hato is a metric of the degree of molecular branching (a lower index indicates a higher degree of branching) and takes the form of:

$$\frac{N}{\sum_{i=1}^{N} \frac{1}{\delta_i}}$$

where N is the total number of atoms and δ_i is the number of bonded partners for atom i.

In [3]:
```
m = checkName("trifluoromethanesulfonate")
mol = Chem.MolFromSmiles(m)
a = Draw.MolToMPL(mol)
deltas=[x.GetDegree() for x in mol.GetAtoms()]
while 0 in deltas:
    deltas.remove(0)
deltas = np.array(deltas, 'd')
nAtoms = mol.GetNumAtoms()
res=nAtoms/sum(1./deltas)
print("{}\t{}".format("bonded neighbors:", deltas))
print("{}\t{}".format("number of atoms:", nAtoms))
print("{}\t\t{}".format("Hato index:", res))
```

```
bonded neighbors:       [ 4.  1.  1.  1.  4.  1.  1.  1.]
number of atoms:        8
Hato index:             1.23076923077
```



We can perform this for all the anions in our salts with the pychem and rdkit modules:

In [4]:
```
from pychem import topology as top

for item in anion_smiles:
    mol = Chem.MolFromSmiles(item)
    print("{}\t{}".format(top.GetTopology(mol)["Hato"],checkName(item)))
```

```
1.176   methanesulfonate
1.231   trifluoromethanesulfonate
1.333   ethyl sulfate
1.176   perchlorate
1.176   tetrafluoroborate
1.271   tris(pentafluoroethyl)trifluorophosphate
1.2     thiocyanate
1.429     dicyanamide
1.448   tricyanomethanide
1.2     acetate
1.313   1,1,2,2,2-pentafluoro-N-[(pentafluoroethyl)sulfonyl]ethanesulfonamide
1.135   hexafluorophosphate
1.304   bis[(trifluoromethyl)sulfonyl]imide
1.263   methylsulfate
1.313   bis(perfluoroethylsulfonyl)imide
1.254   trifluoroacetate
```

Chi4c

back to top

Class: connectivity

This simple index tabulates the occurence of 4th order clusters and normalizes by the product of the number of bonded partners to each atom within the cluster:

$$\sum_{i=1}^{C} \frac{1}{\sqrt{\Pi(\delta_i)}}$$

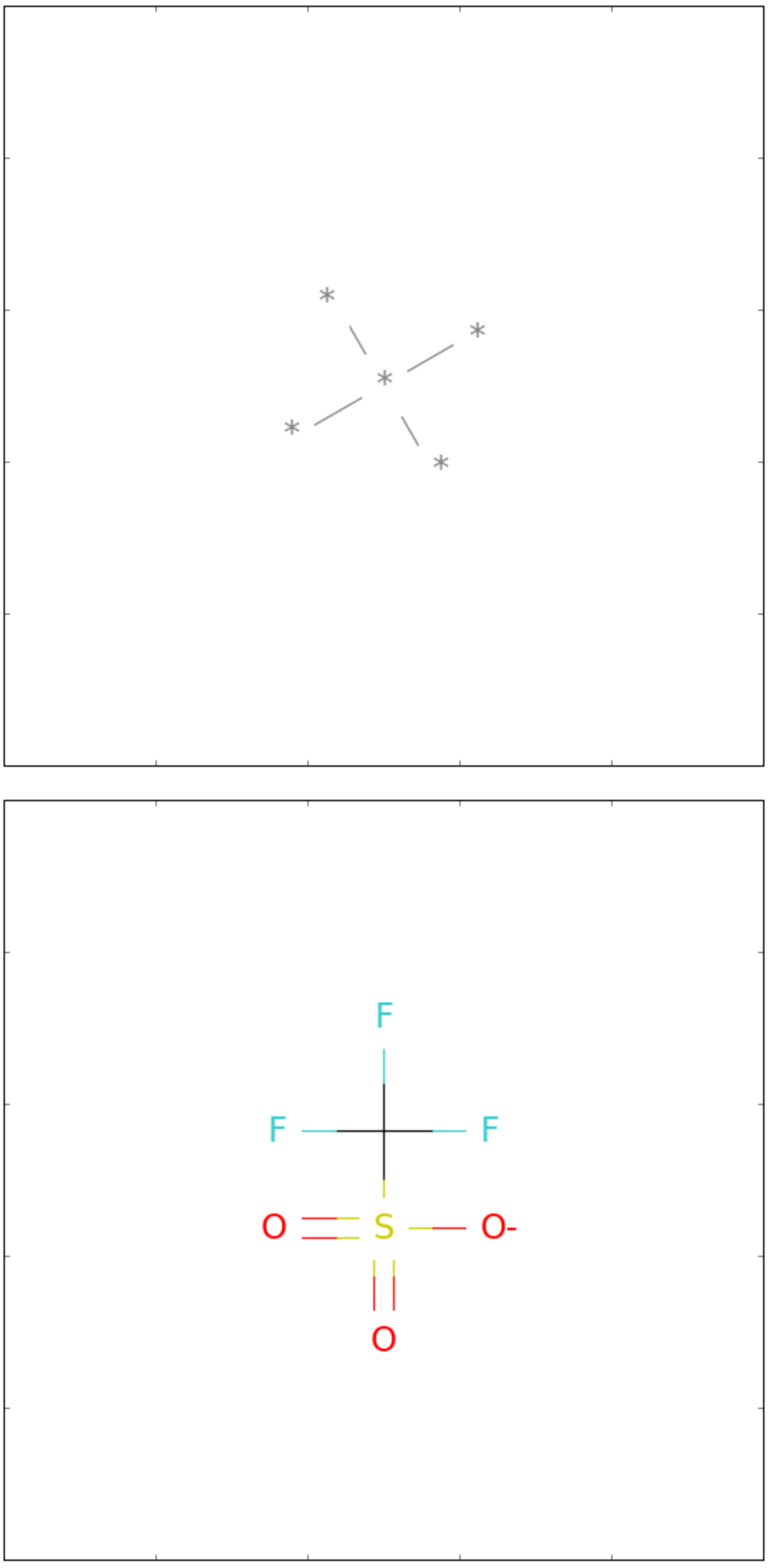
where δ_i is the array of bonded partners for atoms in cluster i and C is the total number of clusters.

In [5]:
```
m = checkName("trifluoromethanesulfonate")
mol = Chem.MolFromSmiles(m)
deltas = [x.GetDegree() for x in mol.GetAtoms()]
names = [x.GetSymbol() for x in mol.GetAtoms()]
patt = Chem.MolFromSmarts('*~*(~*)(~*)~*')
HPatt = mol.GetSubstructMatches(patt)
accum = 0.0
i = 1
for cluster in HPatt:
    deltas = [mol.GetAtomWithIdx(x).GetDegree() for x in cluster]
    while 0 in deltas:
        deltas.remove(0)
    if deltas != []:
        deltas1 = np.array(deltas,np.float).prod()
        accum = accum+1./np.sqrt(deltas1)
        print("{}\t{}\t\t{}\t{}".format("cluster %s:"%i,deltas,"and product:",deltas1))
        i += 1
print("{}\t{}".format("Chi4c index:",accum))
```

```
cluster 1:      [4, 4, 1, 1, 1]         and product:    16.0
cluster 2:      [1, 4, 1, 1, 4]         and product:    16.0
Chi4c index:    0.5
```

We can see how this cluster appears visually within the anion we're investigating:

In [6]:
```
patt_unambig = Chem.MolFromSmarts('**(*)(*)*')
a = Draw.MolToMPL(patt_unambig)
b = Draw.MolToMPL(mol)
```



We can perform this for all the anions in our salts with the pychem and rdkit modules:

In [7]:
```
from pychem import connectivity as con

for item in anion_smiles:
    mol = Chem.MolFromSmiles(item)
    print("{}\t{}".format(con.GetConnectivity(mol)["Chi4c"],checkName(item)))
```

```
0.5     methanesulfonate
0.5     trifluoromethanesulfonate
0.354   ethyl sulfate
0.5     perchlorate
0.5     tetrafluoroborate
2.74    tris(pentafluoroethyl)trifluorophosphate
0.0     thiocyanate
0.0       dicyanamide
0.0     tricyanomethanide
0.0     acetate
1.104   1,1,2,2,2-pentafluoro-N-[(pentafluoroethyl)sulfonyl]ethanesulfonamide
6.124   hexafluorophosphate
0.854   bis[(trifluoromethyl)sulfonyl]imide
0.354   methylsulfate
1.104   bis(perfluoroethylsulfonyl)imide
0.289   trifluoroacetate
```

Smax14

back to top

Class: Electrotopological

This descriptor returns the maximum electrotopological state (E-state) of any carbon with a triple bonded partner. The E-state of an atom is comprised of an intrinsic value I_i and a perturbation term ΔI_i that comes from the topological environment in which the atom resides. For first row atoms in the periodic table the intrinsic value is:

$$(\delta^v + 1)\delta$$

where δ^v and δ are the counts of the valence and sigma electrons, respectively. the influence of other atoms, ΔI_i is given by:

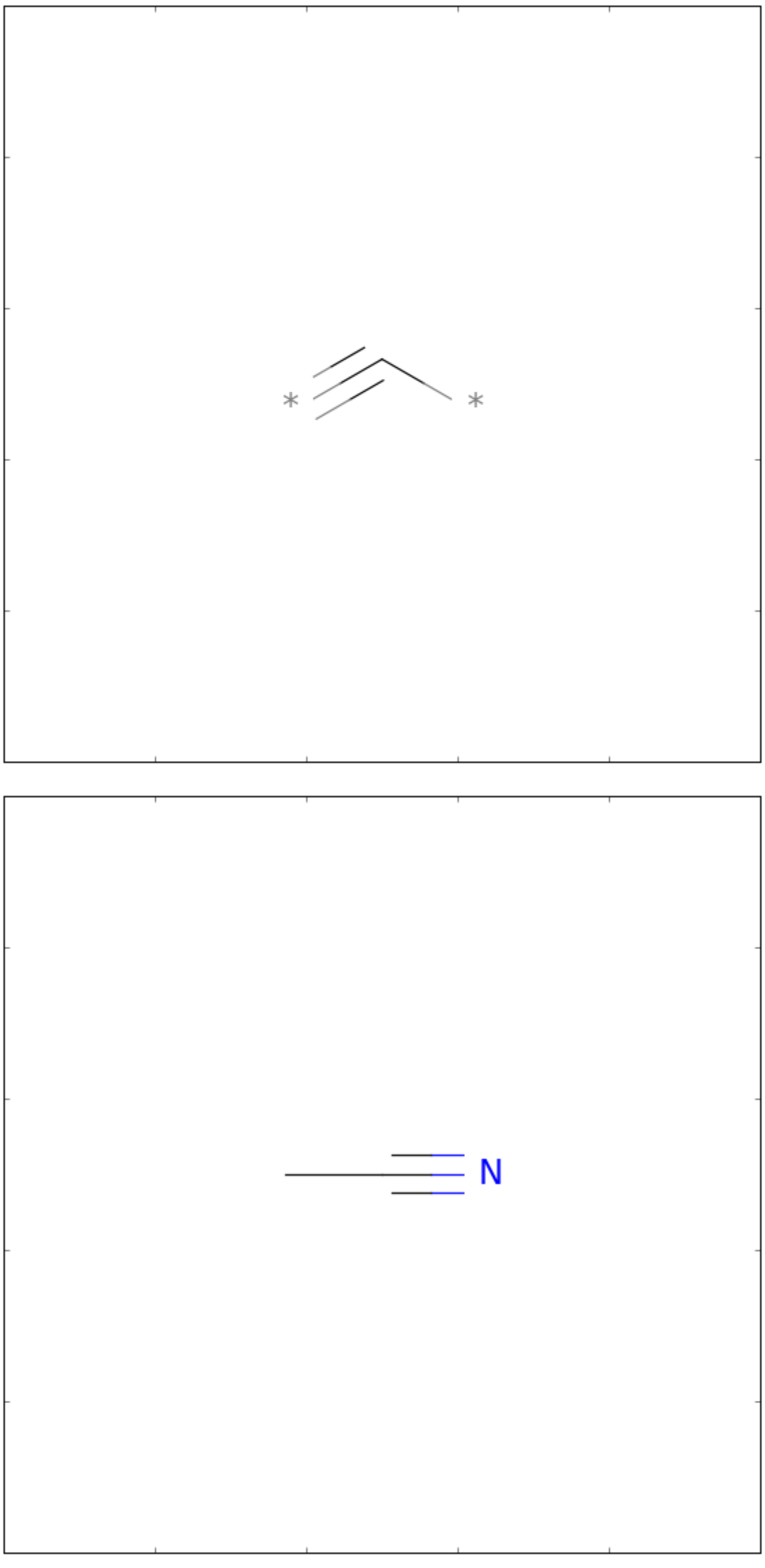
$$\sum \frac{(I_j - Ij)}{r_{ij}^2}$$

where r_{ij} is the topological distance between atoms i and j, counting i and j themselves. The E-state is then given by:

$$I_i + \Delta I_i$$

In [8]:
```
from rdkit.Chem import EState
from pychem.estate import CalculateMaxAtomTypeEState

m = Chem.MolFromSmarts("[CD2HO](#*)-*")
a = Draw.MolToMPL(m)
mol = Chem.MolFromSmiles("N#CC")
b = Draw.MolToMPL(mol)
Ein = EState.EStateIndices(mol)
print("{}\t{}".format("E-states of atoms 1-3:",Ein))
print("{}\t{}".format("E-state of triple bonded C:",\
        CalculateMaxAtomTypeEState(mol)["Smax14"]))
```

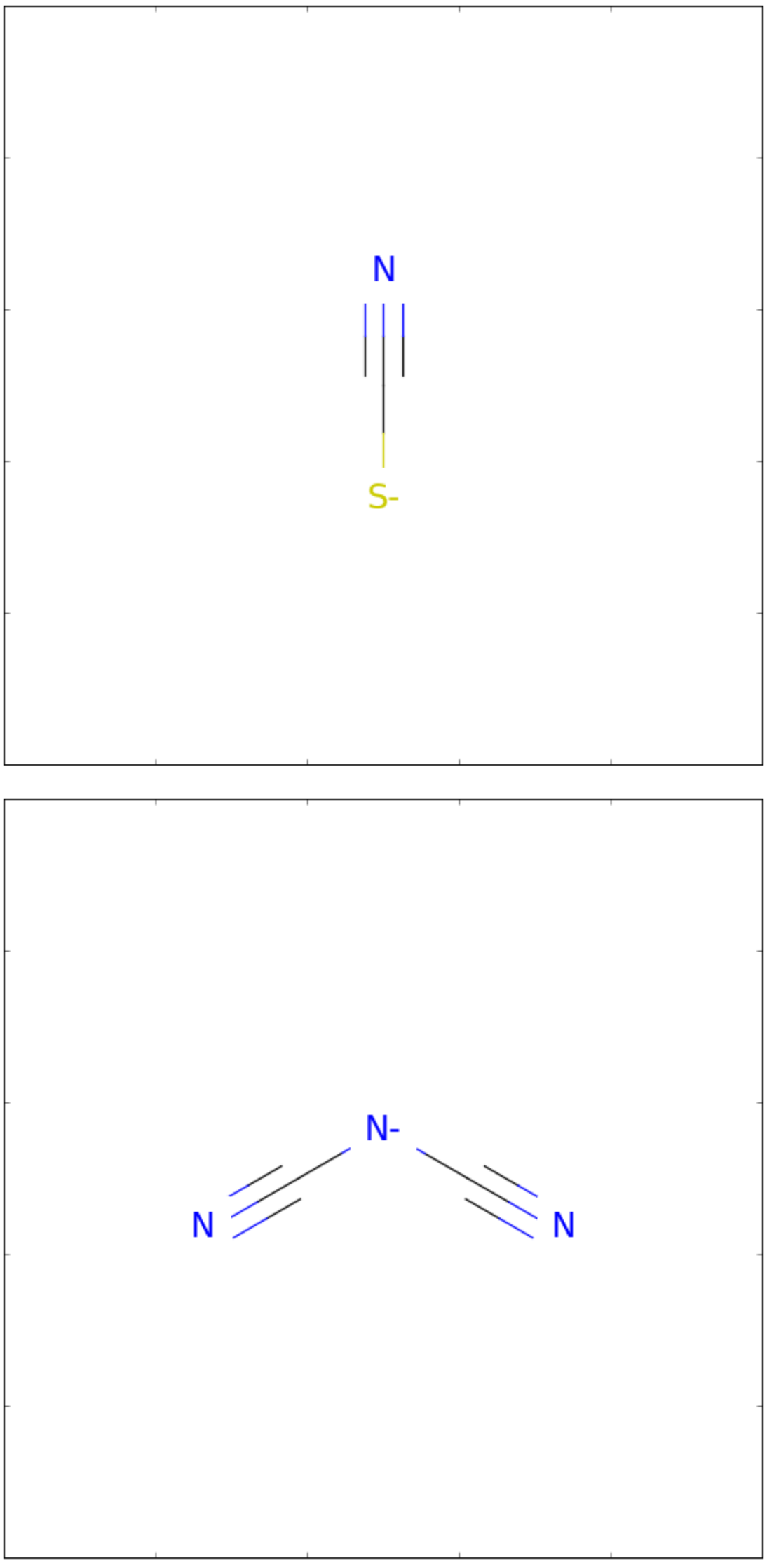
```
E-states of atoms 1-3:  [ 7.31944444  1.75        1.43055556]
E-state of triple bonded C:     1.75
```

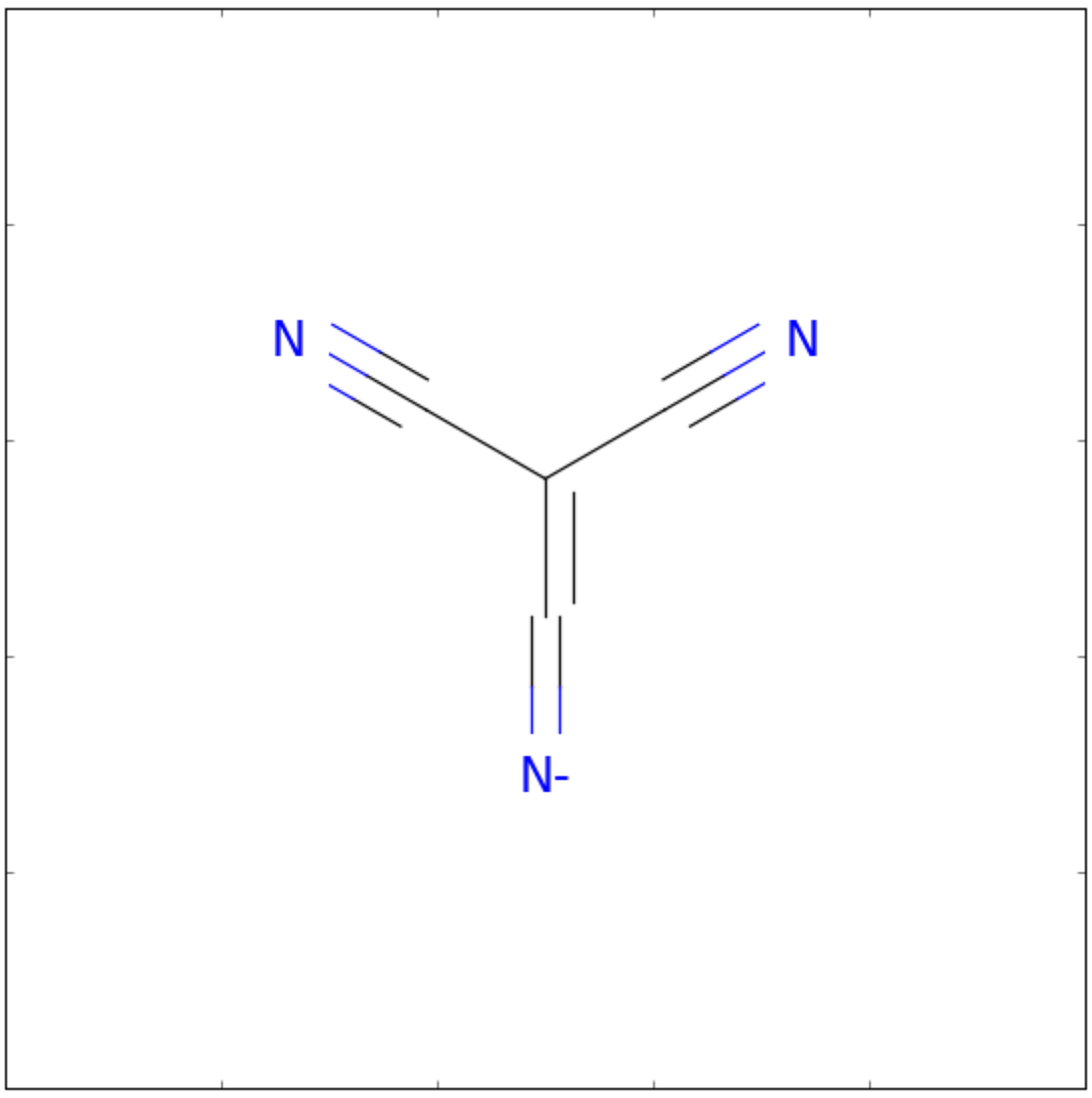


We can do this for all the anions in our dataset:

In [9]:
```
for item in anion_smiles:
    mol = Chem.MolFromSmiles(item)
    if CalculateMaxAtomTypeEState(mol)["Smax14"] != 0:
        print("{}\t{}\n{}\t\t\t{}".format("Max E-State for tsC:",\
            CalculateMaxAtomTypeEState(mol)["Smax14"],"Smiles:",\
            Chem.MolToSmiles(mol)))
        Draw.MolToMPL(mol)
```

Max E-State for tsC: 1.333
Smiles: N#C[S-]
Max E-State for tsC: 1.281
Smiles: N#C[N-]C#N
Max E-State for tsC: 1.396
Smiles: N#CC(=C=[N-])C#N





IC4 & SIC0

back to top

Class: Basak

Basak descriptors contain weighted structural/complimentary information content about fragments within the molecule.

We can use rdkit to loop through all paths of 4-atom length and create an array object with atom ID's in those paths.

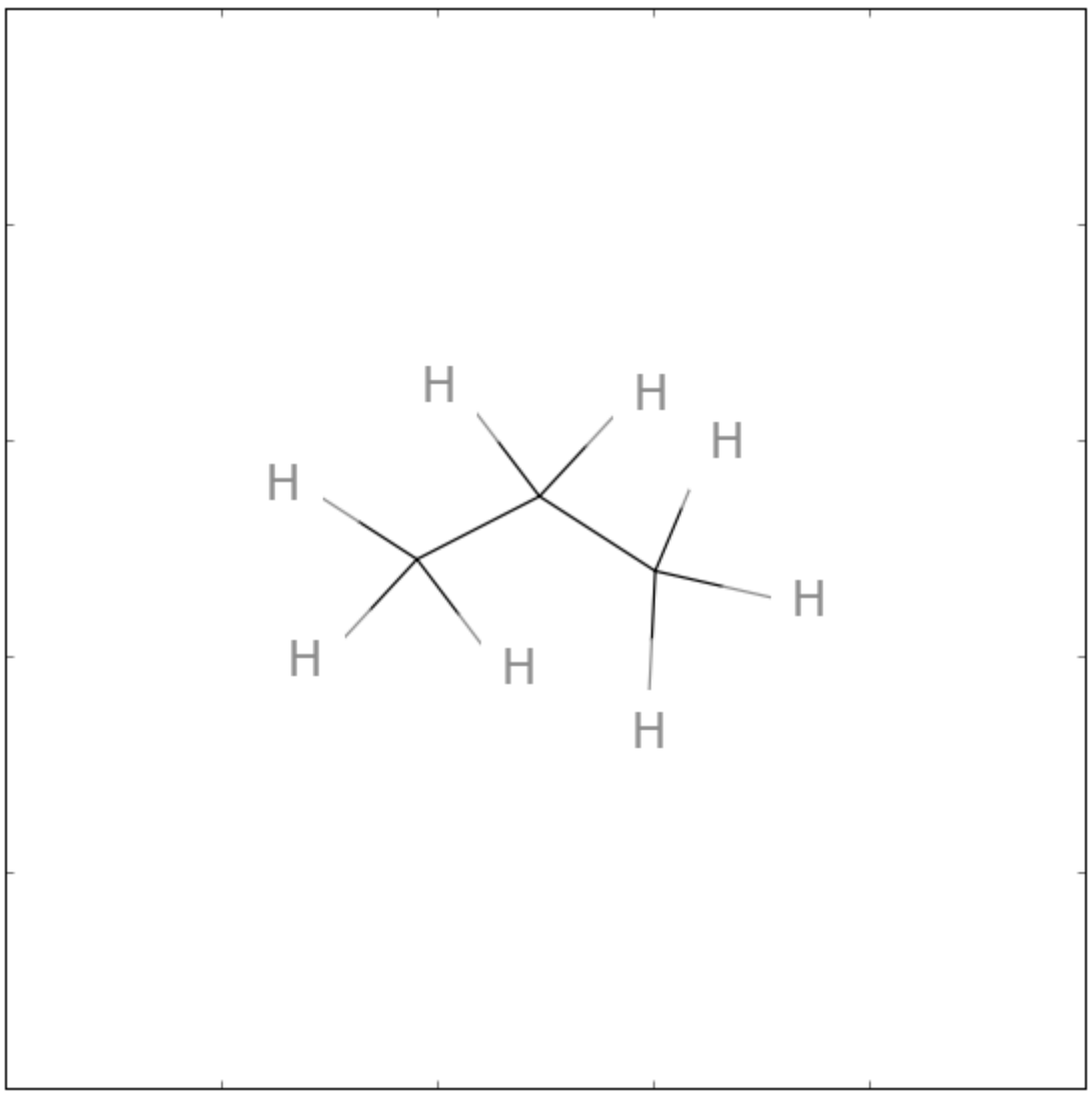
In the following cells we showcase this with butane:

In [10]:
```
import copy
from pychem import basak
from rdkit.Chem import FindAllPathsOfLengthN as path

m = "CCC"
mol = Chem.MolFromSmiles(m)
mol = Chem.AddHs(mol)
nAtoms = mol.GetNumAtoms()
TotalPath = path(mol,4,useBonds=0,useHs=1)
np.array(TotalPath)
```

Out[10]:
```
array([[ 0,  1,  2,  8],
       [ 0,  1,  2,  9],
       [ 0,  1,  2, 10],
       [ 2,  1,  0,  3],
       [ 2,  1,  0,  4],
       [ 2,  1,  0,  5],
       [ 3,  0,  1,  6],
       [ 3,  0,  1,  7],
       [ 4,  0,  1,  6],
       [ 4,  0,  1,  7],
       [ 5,  0,  1,  6],
       [ 5,  0,  1,  7],
       [ 6,  1,  2,  8],
       [ 6,  1,  2,  9],
       [ 6,  1,  2, 10],
       [ 7,  1,  2,  8],
       [ 7,  1,  2,  9],
       [ 7,  1,  2, 10]])
```

At this point, it will also help to take a look at the 2D structure:

In [11]:
```
a = Draw.MolToMPL(mol)
```



Pychem reorganizes that array to create a dictionary object where atom ID entries list the other atom IDs of members in their paths (see output). Notice that the 2 central carbon H's have 6, 4-member paths. These will be distinguished in the IC4 descriptor:

In [12]:
```
IC = {}
for atom in range(nAtoms):
    temp = []
    at = mol.GetAtomWithIdx(atom)
    temp.append(at.GetAtomicNum())
    for index in TotalPath:
        if atom == index[0]:
            temp.append([mol.GetAtomWithIdx(kk).GetAtomicNum() for kk in index[1:]])
        if atom == index[-1]:
            cds = list(index)
            cds.reverse()
            temp.append([mol.GetAtomWithIdx(kk).GetAtomicNum() for kk in cds[1:]])
    IC[str(atom)] = temp
cds=[]
for value in IC.values():
    value.sort()
    cds.append(value)
cds
```

Out[12]:
```
[[1, [6, 6, 1], [6, 6, 1], [6, 6, 6]],
 [6],
 [6, [6, 6, 1], [6, 6, 1], [6, 6, 1]],
 [1, [6, 6, 1], [6, 6, 1], [6, 6, 6]],
 [6, [6, 6, 1], [6, 6, 1], [6, 6, 1]],
 [1, [6, 6, 1], [6, 6, 1], [6, 6, 6]],
 [1, [6, 6, 1], [6, 6, 1], [6, 6, 6]],
 [1, [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1]],
 [1, [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1]],
 [1, [6, 6, 1], [6, 6, 1], [6, 6, 6]],
 [1, [6, 6, 1], [6, 6, 1], [6, 6, 6]]]
```

Now that we have a nested array in the form of:

```
[starting_atom [atom2_in_path, atom3_in_path, ...]

[atom2_in_path, atom3_in_path, ...]],

[starting_atom [ ... ], [ ... ]], .etc... ]
```

we need to tally all fragments that are unique by the atom types in the path, and the number of paths occuring at a given origin. Looking at the structure in `cds` above, we expect to count 4 unique fragments; one with six members, one with two members, one with two members, and one with a single member.

note that the middle carbon atom has no 4-member paths: `[[6]]`

In [13]:
```
cds=[]
for value in IC.values():
    value.sort()
    cds.append(value)

kkk=range(nAtoms)
aaa=copy.deepcopy(kkk)
res=[]
for i in aaa:
    if i in kkk:
        jishu=0
        kong=[]
        temp1=cds[i]
        for j in aaa:
            if cds[j]==temp1:

                jishu=jishu+1
                kong.append(j) #grab atom indices with paths of same atomic numbers
        print("{}\n{}".format("unique fragments found in the form of:",temp1))
        print("{}\n{}\n".format("for atoms:",kong))
        for ks in kong:
            kkk.remove(ks)
        res.append(jishu)

print("{}\n{}\n".format("array of unique fragment counts:",res))
```

```
unique fragments found in the form of:
[1, [6, 6, 1], [6, 6, 1], [6, 6, 6]]
for atoms:
[0, 3, 5, 6, 9, 10]

unique fragments found in the form of:
[6]
for atoms:
[1]

unique fragments found in the form of:
[6, [6, 6, 1], [6, 6, 1], [6, 6, 1]]
for atoms:
[2, 4]

unique fragments found in the form of:
[1, [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1], [6, 6, 1]]
for atoms:
[7, 8]

array of unique fragment counts:
[6, 1, 2, 2]
```

The array of unique fragment counts can then converted into probabilities given the population of unique fragments:

In [14]:
```
probability = np.array(res,np.float)/sum(res)
IC4=0.0
for i in probability:
    if i != 0:
        IC4=IC4-i*np.log2(i)
print(IC4)
```

```
1.68581570915
```

This is all performed in the pychem module:

In [15]:
```
basak._CalculateBasakICn(mol,4)
```

Out[15]:
```
1.6858157091530304
```

We can do this for all the cations in our dataset:

In [16]:
```
import pandas as pd
import os
for item in cation_smiles:
    mol = Chem.MolFromSmiles(item)
    if basak._CalculateBasakICn(mol,4) != 0:
        print("{}\t{}\t{}\t{}".format("IC4:",\
            basak._CalculateBasakICn(mol,4),"Smiles:",\
            Chem.MolToSmiles(mol)))
```

```
IC4:    4.01218840397   Smiles: CCCCn1cc[n+](C)c1C
IC4:    3.8035088548    Smiles: CCCC[n+]1ccccc1
IC4:    4.18045139089   Smiles: CCCC[n+]1cccc(C)c1
IC4:    3.26408351177   Smiles: CCCC[N+]1(C)CCCC1
IC4:    3.9434651896    Smiles: CCCCn1cc[n+](C)c1
IC4:    2.53063906223   Smiles: Cn1cc[n+](C)c1
IC4:    4.16976696234   Smiles: CCCCCCCC[n+]1cccc(C)c1
IC4:    3.97945944659   Smiles: CCCCCCCCn1cc[n+](C)c1
IC4:    4.23140184539   Smiles: CCCC[NH+]1C=CN(C)C1C
IC4:    3.8841550946    Smiles: CCCC[n+]1ccc(C)cc1
IC4:    3.43162356585   Smiles: CCn1cc[n+](C)c1
IC4:    4.06678421347   Smiles: CCCCCCn1cc[n+](C)c1
IC4:    3.14316643316   Smiles: CCCCCCCCCCCCCC[P+](CCCCCC)(CCCCCC)CCCCCC
IC4:    3.83326968952   Smiles: CCCn1cc[n+](C)c1C
```

The structural information content Basak descriptors are a simple variation of the information content (ICn) descriptors:

$$SIC_n = \frac{IC_n}{\log_2(N)}$$

where N is the total number of atoms. The SIC0 descriptor can be calculated as follows:

In [17]:
```
from pychem import basak
mol = Chem.MolFromSmiles("CCC")
Hmol = Chem.AddHs(mol)
nAtoms = Hmol.GetNumAtoms()
IC0 = basak.CalculateBasakIC0(mol)
SIC0 = IC0/np.log2(nAtoms)
SIC0
```

Out[17]:
```
0.24436122167242844
```

We can do this for all the cations in our dataset:

In [18]:
```
import pandas as pd
import os
for item in anion_smiles:
    mol = Chem.MolFromSmiles(item)
    if basak._CalculateBasakICn(mol,4) != 0:
        print("{}\t{}\t{}\t{}".format("SIC0:",\
            basak._CalculateBasakICn(mol,4),"Smiles:",\
            Chem.MolToSmiles(mol)))
```

```
SIC0:   2.41938194565   Smiles: CS(=O)(=O)O
SIC0:   1.81127812446   Smiles: O=S(=O)([O-])C(F)(F)F
SIC0:   2.62581458369   Smiles: CCOS(=O)(=O)[O-]
SIC0:   1.79248125036   Smiles: [O-][Cl+3]([O-])([O-])O
SIC0:   2.31170568882   Smiles: FC(F)(F)C(F)(F)[P-](F)(F)(F)(C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
SIC0:   1.52192809489   Smiles: N#C[N-]C#N
SIC0:   1.44881563573   Smiles: N#CC(=C=[N-])C#N
SIC0:   1.55665670746   Smiles: CC(=O)[O-]
SIC0:   2.75444184571   Smiles: O=S(=O)(NS(=O)(=O)C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
SIC0:   2.07290559532   Smiles: O=S(=O)([N-]S(=O)(=O)C(F)(F)F)C(F)(F)F
SIC0:   2.11328333429   Smiles: COS(=O)(=O)[O-]
SIC0:   2.60613766067   Smiles: O=S(=O)([N-]S(=O)(=O)C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
SIC0:   1.55665670746   Smiles: O=C([O-])C(F)(F)F
```

EstateVSA3, PEOEVSA6, & PEOEVSA12

back to top

Class: Molecular Operating Environment (MOE)

The MOE-type descriptors bin the output from other descriptor types and calculate the van der Waals (VDWs) surface area (VSA) of atoms contributing to any specified bin of that output. EstateVSA3 calculates the sum of VSA contributions to electrotopological states within 0.717-1.165. The VSA is calculated as:

$$4\pi R_i^2 - \pi R_i^2 \sum_{j=1}^{N} a_{ij}\left(\frac{R_j^2 - (R_i - g_{ij})^2}{g_{ij}}\right)$$

where R is the VDWs radius, N is for all atoms bonded to the ith atom, and α_{ij} are the elements of the adjacency matrix. g_{ij} is calculated as:

$$min(max(|R_i - R_j|, b_{ij}), (R_i + R_j))$$

where b_{ij} is the ideal bond length formed by atoms i and j, calculated as:

$$r_i + r_j - c_{ij}$$

where c_{ij} is a correction term for multiplicity (0 for single bond, 0.1 for aromatic bond, 0.2 for double bond, and 0.3 for triple bonds).

The E-state is calculated as described earlier.

In [19]:
```
from pychem import moe
m = "CC(=O)[O-]"
mol = Chem.MolFromSmiles(m)
moe.CalculateVSAEstate(mol)
print("{}\t{}".format("VSA of C,C,O,O:",mol._labuteContribs[1:]))
```

```
VSA of C,C,O,O: [6.923737199690624, 5.969305287951849, 4.794537184071822, 5.106527394840706]
```

Pychem computes the VSA contribution to all the binned Estates. Default bins are -0.39,0.29,0.717,1.165,1.54,1.807,2.05,4.69,9.17,15. We see that the outer carbon is soley contributing to Estates in the 0.717-1.165 bin range

In [20]:
```
moe.CalculateEstateVSA(mol)
```

Out[20]:
```
{'EstateVSA0': 5.969,
 'EstateVSA1': 0.0,
 'EstateVSA10': 0.0,
 'EstateVSA2': 0.0,
 'EstateVSA3': 6.924,
 'EstateVSA4': 0.0,
 'EstateVSA5': 0.0,
 'EstateVSA6': 0.0,
 'EstateVSA7': 0.0,
 'EstateVSA8': 9.901,
 'EstateVSA9': 0.0}
```

We can do this for all the cations in our dataset:

In [21]:
```
import pandas as pd
import os
from pychem import moe
for item in anion_smiles:
    mol = Chem.MolFromSmiles(item)
    if basak._CalculateBasakICn(mol,4) != 0:
        print("{}\t{}\t{}\t{}".format("EstateVSA3:",\
            moe.CalculateEstateVSA(mol)["EstateVSA3"],"Smiles:",\
            Chem.MolToSmiles(mol)))
```

```
EstateVSA3:     0.0     Smiles: CS(=O)(=O)O
EstateVSA3:     0.0     Smiles: O=S(=O)([O-])C(F)(F)F
EstateVSA3:     0.0     Smiles: CCOS(=O)(=O)[O-]
EstateVSA3:     0.0     Smiles: [O-][Cl+3]([O-])([O-])O
EstateVSA3:     0.0     Smiles: FC(F)(F)C(F)(F)[P-](F)(F)(F)(C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
EstateVSA3:     0.0     Smiles: N#C[N-]C#N
EstateVSA3:     0.0     Smiles: N#CC(=C=[N-])C#N
EstateVSA3:     6.924   Smiles: CC(=O)[O-]
EstateVSA3:     0.0     Smiles: O=S(=O)(NS(=O)(=O)C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
EstateVSA3:     4.127   Smiles: O=S(=O)([N-]S(=O)(=O)C(F)(F)F)C(F)(F)F
EstateVSA3:     7.11    Smiles: COS(=O)(=O)[O-]
EstateVSA3:     0.0     Smiles: O=S(=O)([N-]S(=O)(=O)C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
EstateVSA3:     0.0     Smiles: O=C([O-])C(F)(F)F
```

As with the EstateVSA MOE descriptor, the PEOEVSA MOE variety calculate the VSA of atoms contributing to a specified bin of partial charge. PEOEVSA6 and PEOEVSA12 calculate the sum of VSA contributions to partial charges within -0.05-0.0 and 0.23-0.30, respectively. The VSA is calculated in the same way as before. The partial charges are calculated by the Gasteiger method.

In [22]:
```
from pychem import moe
m = "CCOS(=O)(=O)[O-]"
mol = Chem.MolFromSmiles(m)
moe.CalculateVSAEstate(mol)
print("{}\t{}".format("VSAs:",mol._labuteContribs[1:]))
```

```
VSAs:   [6.923737199690624, 6.606881964512918, 4.183085432649707, 10.399000581649606, 4.208898492164
469, 4.208898492164469, 4.552749873690364]
```

In [23]:
```
#pychem computes the VSA contribution to all the binned partial charges
#default bins are [-.3,-.25,-.20,-.15,-.10,-.05,0,.05,.10,.15,.20,.25,.30]
#we see that the outer carbon is soley contributing to partial charge in the
#-0.05-0.0 bin range
moe.CalculatePEOEVSA(mol)
```

Out[23]:
```
{'PEOEVSA0': 4.553,
 'PEOEVSA1': 4.183,
 'PEOEVSA10': 0.0,
 'PEOEVSA11': 10.399,
 'PEOEVSA12': 0.0,
 'PEOEVSA13': 0.0,
 'PEOEVSA2': 8.418,
 'PEOEVSA3': 0.0,
 'PEOEVSA4': 0.0,
 'PEOEVSA5': 0.0,
 'PEOEVSA6': 6.924,
 'PEOEVSA7': 0.0,
 'PEOEVSA8': 6.607,
 'PEOEVSA9': 0.0}
```

We can do this for all ions in the dataset:

In [24]:
```
import os
from pychem import moe
smiles=set()
for item in anion_smiles:
    mol = Chem.MolFromSmiles(item)
    if basak._CalculateBasakICn(mol,4) != 0:
        print("{}\t{}\t{}\t{}".format("PEOEVSA6:",\
            moe.CalculatePEOEVSA(mol)["PEOEVSA6"],"Smiles:",\
            Chem.MolToSmiles(mol)))
for item in cation_smiles:
    mol = Chem.MolFromSmiles(item)
    if basak._CalculateBasakICn(mol,4) != 0:
        print("{}\t{}\t{}\t{}".format("PEOEVSA12:",\
            moe.CalculatePEOEVSA(mol)["PEOEVSA12"],"Smiles:",\
            Chem.MolToSmiles(mol)))
```

```
PEOEVSA6:       0.0     Smiles: CS(=O)(=O)O
PEOEVSA6:       0.0     Smiles: O=S(=O)([O-])C(F)(F)F
PEOEVSA6:       6.924   Smiles: CCOS(=O)(=O)[O-]
PEOEVSA6:       0.0     Smiles: [O-][Cl+3]([O-])([O-])O
PEOEVSA6:       0.0     Smiles: FC(F)(F)C(F)(F)[P-](F)(F)(F)(C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
PEOEVSA6:       0.0     Smiles: N#C[N-]C#N
PEOEVSA6:       0.0     Smiles: N#CC(=C=[N-])C#N
PEOEVSA6:       6.924   Smiles: CC(=O)[O-]
PEOEVSA6:       0.0     Smiles: O=S(=O)(NS(=O)(=O)C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
PEOEVSA6:       0.0     Smiles: O=S(=O)([N-]S(=O)(=O)C(F)(F)F)C(F)(F)F
PEOEVSA6:       0.0     Smiles: COS(=O)(=O)[O-]
PEOEVSA6:       0.0     Smiles: O=S(=O)([N-]S(=O)(=O)C(F)(F)C(F)(F)F)C(F)(F)C(F)(F)F
PEOEVSA6:       0.0     Smiles: O=C([O-])C(F)(F)F
PEOEVSA12:      5.824   Smiles: CCCCn1cc[n+](C)c1C
PEOEVSA12:      0.0     Smiles: CCCC[n+]1ccccc1
PEOEVSA12:      0.0     Smiles: CCCC[n+]1cccc(C)c1
PEOEVSA12:      0.0     Smiles: CCCC[N+]1(C)CCCC1
PEOEVSA12:      0.0     Smiles: CCCCn1cc[n+](C)c1
PEOEVSA12:      0.0     Smiles: Cn1cc[n+](C)c1
PEOEVSA12:      0.0     Smiles: CCCCCCCC[n+]1cccc(C)c1
PEOEVSA12:      0.0     Smiles: CCCCCCCCn1cc[n+](C)c1
PEOEVSA12:      0.0     Smiles: CCCC[NH+]1C=CN(C)C1C
PEOEVSA12:      0.0     Smiles: CCCC[n+]1ccc(C)cc1
PEOEVSA12:      0.0     Smiles: CCn1cc[n+](C)c1
PEOEVSA12:      0.0     Smiles: CCCCCCn1cc[n+](C)c1
PEOEVSA12:      0.0     Smiles: CCCCCCCCCCCCCC[P+](CCCCCC)(CCCCCC)CCCCCC
PEOEVSA12:      5.824   Smiles: CCCn1cc[n+](C)c1C
```

GATSp1, MATSe5, MATSp5, bcute2, & bcute5

back to top

Class: Autocorrelation

Spatial autocorrelations, in the case of our models, are distinguised by: 1) the form of the autocorrelation (Moran, Burden, or Geary), 2) the atomic property being correlated (atomic massses, VDWs volumes, Sanderson electronegativities, atomic electronegativities, and polarizabilities), 3) and the topological distance of the autocorrelation.

As an example, MATSe5 or Moran's spatial autocorrelation of sanderson electronegativity with topological distance of five takes on the functional form:

$$\frac{\frac{1}{\Delta}\sum_{i=1}^{N}\sum_{j}^{N}\delta_{i,j}(w_i-\bar{w})(w_j-\bar{w})}{\frac{1}{N}\sum_{i=1}^{N}(w_i-\bar{w})}$$

where N is the total number of atoms, j is incremented at intervals of 5 starting from i along the molecular structure, w_i and w_j are the atomic property evaluated at the current atom, and $\bar{w}$ is the average of the atomic property for the entire molecule.

In [25]:
```
from pychem import moran
mol = Chem.MolFromSmiles("CCCC[n+]1ccccc1")
moran._CalculateMoranAutocorrelation(mol,lag=5,propertylabel='En')
```

Out[25]: 0.111

In [26]:
```
import pandas as pd
import os
from pychem import moran
for item in cation_smiles:
    mol = Chem.MolFromSmiles(item)
    if moran._CalculateMoranAutocorrelation(mol,lag=5) != 0:
        print("{}\t{}\t{}\t{}".format("MATSe5:",\
            moran._CalculateMoranAutocorrelation(mol,lag=5),"Smiles:",\
            Chem.MolToSmiles(mol)))
```

```
MATSe5: 0.019   Smiles: CCCCn1cc[n+](C)c1C
MATSe5: 0.111   Smiles: CCCC[n+]1ccccc1
MATSe5: 0.1     Smiles: CCCC[n+]1cccc(C)c1
MATSe5: 0.111   Smiles: CCCC[N+]1(C)CCCC1
MATSe5: -0.036  Smiles: CCCCCCCC[n+]1cccc(C)c1
MATSe5: -0.093  Smiles: CCCCCCCCn1cc[n+](C)c1
MATSe5: 0.019   Smiles: CCCC[NH+]1C=CN(C)C1C
MATSe5: 0.1     Smiles: CCCC[n+]1ccc(C)cc1
MATSe5: 0.333   Smiles: CCn1cc[n+](C)c1
MATSe5: -0.143  Smiles: CCCCCCn1cc[n+](C)c1
MATSe5: -0.072  Smiles: CCCCCCCCCCCCCC[P+](CCCCCC)(CCCCCC)CCCCCC
MATSe5: -0.063  Smiles: CCCn1cc[n+](C)c1C
```

Models

In the following section we showcase the general methodology by which we produce our models. It should be noted that the following cells are meant as a demonstration only. The reader will note the omission of certain details including: the many methods of parameterizing LASSO, the creation of category based subsets of the salt data, the evaluation of t-scores and p-values, and feature coefficient vs lambda plots. These are omitted in the interest of succinctness and are available upon request to the authors.

LASSO

back to top

We shrink the feature space before feeding it into a neural network.

Scikit-learn has a random search algorithm that is useful and fairly easy to implement. In our work we used bootstrap, cross validation, and shuffle-split to parameterize LASSO on the ILThermo viscosity data.

In [7]:
```
import pandas as pd
import numpy as np
from math import log
from scipy.stats import uniform as sp_rand
from sklearn.linear_model import Lasso
from sklearn.model_selection import RandomizedSearchCV

df = pd.DataFrame.from_csv('viscosity_processed.csv', index_col=None)
df["Viscosity"] = df["Viscosity"].apply(lambda x: log(float(x)))
metaDf = df.select_dtypes(include=["object"])
dataDf = df.select_dtypes(include=[np.number])
X_train = dataDf.values[:,:-1]
Y_train = dataDf.values[:,-1]

#parameterize the LASSO
param_grid = {"alpha": sp_rand(0.001,0.01)}
model = Lasso(max_iter=1e6,tol=1e-6)
grid = RandomizedSearchCV(estimator=model, param_distributions=param_grid, n_jobs=-1,\
                          n_iter=15)
grid_result = grid.fit(X_train, Y_train)
print(grid_result.best_estimator_)
```

```
Lasso(alpha=0.010504530395576327, copy_X=True, fit_intercept=True,
   max_iter=1000000.0, normalize=False, positive=False, precompute=False,
   random_state=None, selection='cyclic', tol=1e-06, warm_start=False)
```

Confidence Intervals

back to top

It can be incredibly useful to look at our coefficient response to changes in the underlying training data (e.g. does it look like one of our features are being selected because of a single type of training datum, category of salt, etc.)

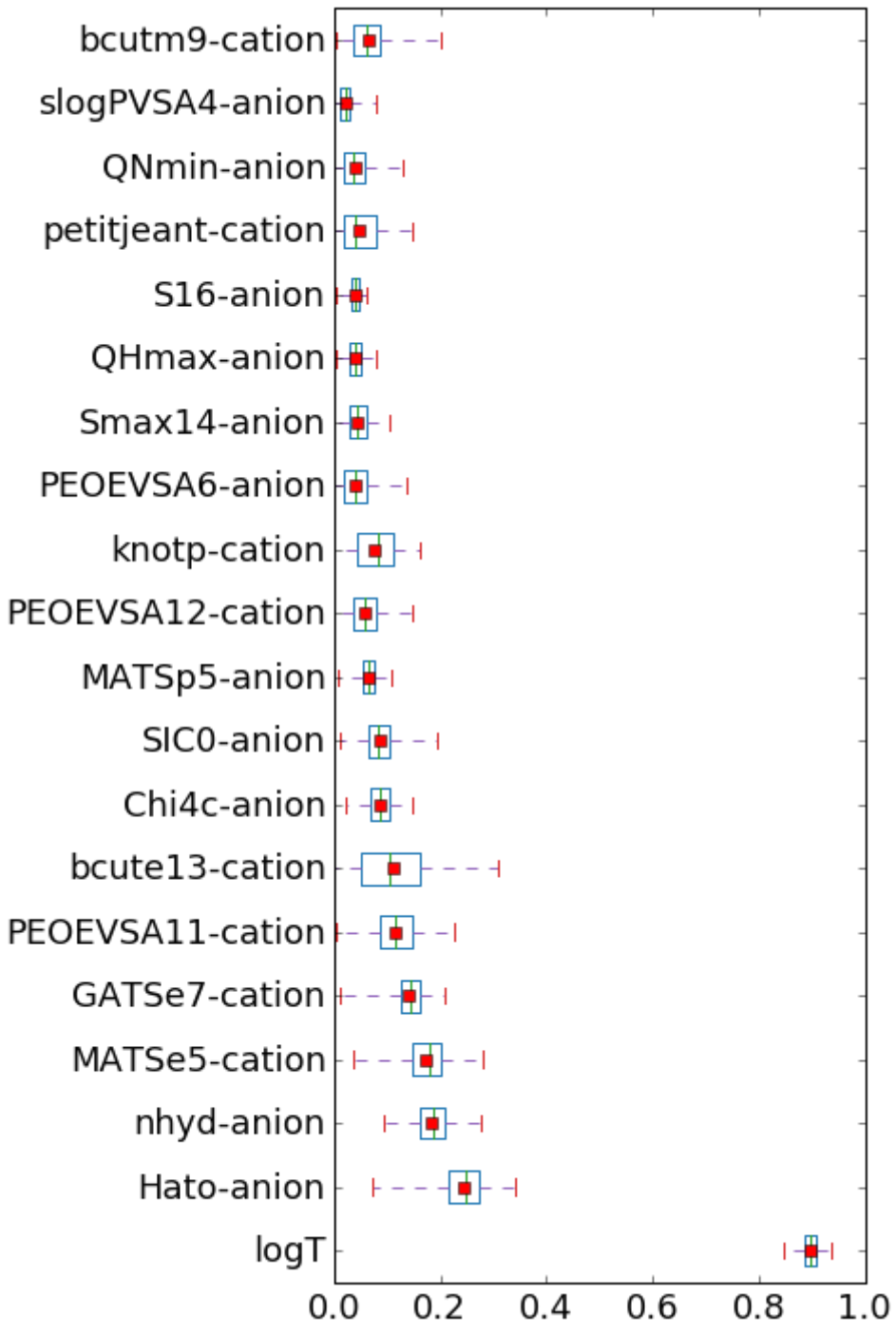
We assess this using the bootstrap.

In [10]:
```
from numpy.random import randint

iterations=300
averages=np.zeros(iterations)
variances=np.zeros(iterations)
test_MSE_array=[]

metadf = df.select_dtypes(include=["object"])
datadf = df.select_dtypes(include=[np.number])


data=np.array(datadf)
n = data.shape[0]
d = data.shape[1]
d -= 1
n_train = int(n*0.8) #set fraction of data to be for training
n_test  = n - n_train
deslist=datadf.columns
score=np.zeros(len(datadf.columns))
feature_coefficients=np.zeros((len(datadf.columns),iterations))
test_MSE_array=[]
model_intercept_array=[]
for i in range(iterations):
    data = np.random.permutation(data)
    X_train = np.zeros((n_train,d)) #prepare train/test arrays
    X_test  = np.zeros((n_test,d))
    Y_train = np.zeros((n_train))
    Y_test = np.zeros((n_test))

    ###sample from training set with replacement
    for k in range(n_train):
        x = randint(0,n_train)
        X_train[k] = data[x,:-1]
        Y_train[k] = (float(data[x,-1]))
    n = data.shape[0]
    ###sample from test set with replacement
    for k in range(n_test):
        x = randint(n_train,n)
        X_test[k] = data[x,:-1]
        Y_test[k] = (float(data[x,-1]))

    ###train the lasso model
    model = Lasso(alpha=0.010930152533240051,tol=1e-10,max_iter=4000)
    model.fit(X_train,Y_train)

    ###Check what features are selected
    p=0
    avg_size=[]
    for a in range(len(data[0])-1):
        if model.coef_[a] != 0:
            score[a] = score[a] + 1
            feature_coefficients[a,i] = model.coef_[a] ###append the model coefs
            p+=1
    avg_size.append(p)

    ###Calculate the test set MSE
    Y_hat = model.predict(X_test)
    n = len(Y_test)
    test_MSE = np.sum((Y_test-Y_hat)**2)**1/n
    test_MSE_array.append(test_MSE)

    ###Grab intercepts
    model_intercept_array.append(model.intercept_)

print("{}\t{}".format("average feature length:", np.average(avg_size)))
print("{}\t{}".format("average y-intercept:", "%.2f" % np.average(model_intercept_array)))
print("{}\t{}".format("average test MSE:", "%.2E" % np.average(test_MSE_array)))
print("{}\t{}".format("average MSE std dev:", "%.2E" % np.std(test_MSE_array)))
select_score=[]
select_deslist=[]
feature_coefficient_averages=[]
feature_coefficient_variance=[]
feature_coefficients_all=[]
for a in range(len(deslist)):
    if score[a] != 0:
        select_score.append(score[a])
        select_deslist.append(deslist[a])
        feature_coefficient_averages.append(np.average(feature_coefficients[a,:]))
        feature_coefficient_variance.append(np.std(feature_coefficients[a,:]))
        feature_coefficients_all.append(feature_coefficients[a,:])
```

```
average feature length: 33.0
average y-intercept:    -3.01
average test MSE:       8.14E-02
average MSE std dev:    1.73E-02
```

In [13]:
```
import matplotlib.pyplot as plt
%matplotlib inline
tableau20 = [(31, 119, 180), (174, 199, 232), (255, 127, 14), (255, 187, 120),
             (44, 160, 44), (152, 223, 138), (214, 39, 40), (255, 152, 150),
             (148, 103, 189), (197, 176, 213), (140, 86, 75), (196, 156, 148),
             (227, 119, 194), (247, 182, 210), (127, 127, 127), (199, 199, 199),
             (188, 189, 34), (219, 219, 141), (23, 190, 207), (158, 218, 229)]

# Scale the RGB values to the [0, 1] range, which is the format matplotlib accepts.
for i in range(len(tableau20)):
    r, g, b = tableau20[i]
    tableau20[i] = (r / 255., g / 255., b / 255.)

#save the selected feature coeffs and their scores
df = pd.DataFrame(select_score, select_deslist)
bootstrap_list_scores = df

#save the selected feature coefficients
df = pd.DataFrame(data=np.array(feature_coefficients_all).T, columns=select_deslist)
df = df.T.sort_values(by=1, ascending=False)
bootstrap_coefficients = df

#save all the bootstrap data to create a box & whiskers plot
df = pd.DataFrame(data=[feature_coefficient_averages,\
                  feature_coefficient_variance], columns=select_deslist)
df = df.T.sort_values(by=1, ascending=False)
bootstrap_coefficient_estimates = df

#save the coefficients sorted by their abs() values
df = pd.DataFrame(select_score, select_deslist)
df = df.sort_values(by=0, ascending=False).iloc[:]
cols = df.T.columns.tolist()
df = bootstrap_coefficient_estimates
df = df.loc[cols]
med = df.T.median()
med.sort_values()
newdf = df.T[med.index]
bootstrap_coefficient_estimates_top_sorted = newdf

#create box-whiskers plot
model = bootstrap_coefficient_estimates_top_sorted
model2 = model.abs()
df = model2.T.sort_values(by=0, ascending=False).iloc[:20]
cols = df.T.columns.tolist()
df = bootstrap_coefficients
df = df.loc[cols]
med = df.T.median()
med.sort_values()
newdf = df.T[med.index]
newdf = newdf.replace(0, np.nan)
props = dict(boxes=tableau20[0], whiskers=tableau20[8], medians=tableau20[4],\
             caps=tableau20[6])
newdf.abs().plot(kind='box', figsize=(5,12), subplots=False, fontsize=18,\
        showmeans=True, logy=False, sharey=True, sharex=True, whis='range', showfliers=False,\
        color=props,  vert=False)
plt.xticks(np.arange(0,1.1,0.2))
print(df.shape)
```

(20, 300)



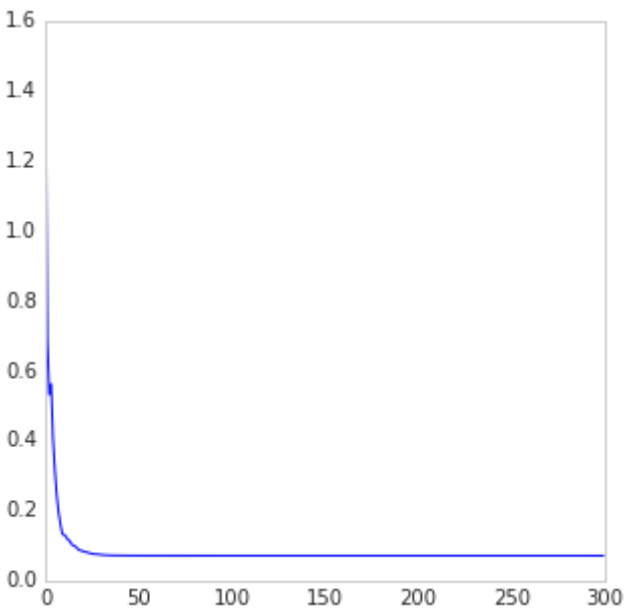
Create Models Progressively Dropping Features

back to top

A last check we performed was to progressively drop features from the LASSO model (based on their average coefficients--see box and whiskers plot above). At some point we should see that the inclusion of additional features doesn't improve the performance of the model. In this case we see improvement fall off at about 10-15 features.

In [16]:
```
mse_scores=[]
for i in range(df.shape[0]):
    model = bootstrap_coefficient_estimates_top_sorted
    model2 = model.abs()
    df = model2.T.sort_values(by=0, ascending=False).iloc[:i]
    cols = df.T.columns.tolist()
    model = model[cols]
    cols = model.columns.tolist()
    cols.append("Viscosity")

    df = pd.DataFrame.from_csv('viscosity_processed.csv', index_col=None)
    df["Viscosity"] = df["Viscosity"].apply(lambda x: log(float(x)))

    df = df.sample(frac=1)
    metadf = df.select_dtypes(include=["object"])
    datadf = df.select_dtypes(include=[np.number])
    df = datadf.T.loc[cols]
    data=np.array(df.T)

    n = data.shape[0]
    d = data.shape[1]
    d -= 1
    n_train = 0#int(n*0.8) #set fraction of data to be for training
    n_test  = n - n_train

    X_train = np.zeros((n_train,d)) #prepare train/test arrays
    X_test  = np.zeros((n_test,d))
    Y_train = np.zeros((n_train))
    Y_test = np.zeros((n_test))
    X_train[:] = data[:n_train,:-1] #fill arrays according to train/test split
    Y_train[:] = (data[:n_train,-1].astype(float))
    X_test[:] = data[n_train:,:-1]
    Y_test[:] = (data[n_train:,-1].astype(float))
    Y_hat = np.dot(X_test, model.loc[0])+np.mean(Y_test[:] - np.dot(X_test[:], model.loc[0]))
    n = len(Y_test)
    test_MSE = np.sum((Y_test-Y_hat)**2)**1/n
    mse_scores.append(test_MSE)
```

In [17]:
```
with plt.style.context('seaborn-whitegrid'):
    fig=plt.figure(figsize=(5,5), dpi=300)
    ax=fig.add_subplot(111)
    ax.plot(mse_scores)
    ax.grid(False)
```



MLPRegressor

back to top

We set avg_selected_features to the number of features we want to include based on the box-whiskers plot, t-tests, and progressively dropped features model. I've set this value to 20 in the cell bellow.

In [24]:
```
####Create dataset according to LASSO selected features
df = bootstrap_list_scores
df = df.sort_values(by=0, ascending=False)
avg_selected_features=20
df = df.iloc[:avg_selected_features]

rawdf = pd.DataFrame.from_csv('viscosity_processed.csv', index_col=None)
rawdf["Viscosity"] = rawdf["Viscosity"].apply(lambda x: log(float(x)))

rawdf = rawdf.sample(frac=1)
metadf = rawdf.select_dtypes(include=["object"])
datadf = rawdf.select_dtypes(include=[np.number])
to_add=[]
for i in range(len(df)):
    to_add.append(df.index[i])
cols = [col for col in datadf.columns if col in to_add]
cols.append("Viscosity")
df = datadf.T.loc[cols]
df = df.T

data=np.array(df)

n = data.shape[0]
d = data.shape[1]
d -= 1
n_train = int(n*0.8) #set fraction of data to be for training
n_test  = n - n_train

X_train = np.zeros((n_train,d)) #prepare train/test arrays
X_test  = np.zeros((n_test,d))
Y_train = np.zeros((n_train))
Y_test = np.zeros((n_test))
X_train[:] = data[:n_train,:-1] #fill arrays according to train/test split
Y_train[:] = (data[:n_train,-1].astype(float))
X_test[:] = data[n_train:,:-1]
Y_test[:] = (data[n_train:,-1].astype(float))
```

We optimize the MLP regressor hyper parameters with any new type of dataset using a random search algorithm. The following script is sent to a high-performance computer:

```
###Randomized Search NN Characterization
param_grid = {"activation": ["identity", "logistic", "tanh", "relu"],\
              "solver": ["lbfgs", "sgd", "adam"], "alpha": sp_rand(),\
              "learning_rate" :["constant", "invscaling", "adaptive"],\
              "hidden_layer_sizes": [randint(100)]}

model = MLPRegressor(max_iter=400,tol=1e-8)

grid = RandomizedSearchCV(estimator=model, param_distributions=param_grid,\
                          n_jobs=-1, n_iter=10)
grid_result = grid.fit(X_train, Y_train)

print(grid_result.best_estimator_)
```

In [26]:
```
from sklearn.neural_network import MLPRegressor

###optimization w/ 100 nodes in hidden layer
model = MLPRegressor(activation='tanh', alpha=0.1993933915059608, batch_size='auto',
       beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=100, learning_rate='adaptive',
       learning_rate_init=0.001, max_iter=100000000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=None,
       shuffle=True, solver='lbfgs', tol=1e-08, validation_fraction=0.1,
       verbose=False, warm_start=False)

model.fit(X_train,Y_train)
```

Out[26]:
```
MLPRegressor(activation='tanh', alpha=0.199393391506, batch_size='auto',
       beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=100, learning_rate='adaptive',
       learning_rate_init=0.001, max_iter=100000000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=None,
       shuffle=True, solver='lbfgs', tol=1e-08, validation_fraction=0.1,
       verbose=False, warm_start=False)
```

In [27]:
```
with plt.style.context('seaborn-whitegrid'):
    fig=plt.figure(figsize=(6,6), dpi=300)
    ax=fig.add_subplot(111)
    ax.plot(np.exp(Y_test),np.exp(model.predict(X_test)),\
            marker=".",linestyle="")
```