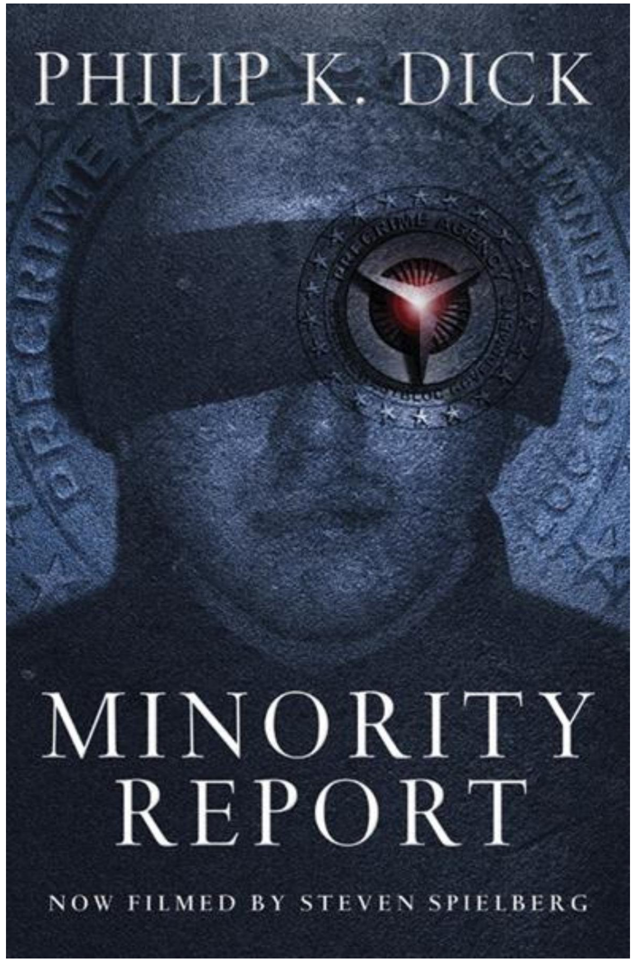
Fairness in Machine Learning

Fairness Issues in Machine Learning

Say we want to estimate the risk of violent crimes in given population



- This is obviously a very ethically sensitive (and questionable) task
- ...Since our model may easily end up discriminating some social groups

This makes it a good test case to discuss fairness in data-driven methods

Fairness in Data-Driven Methods

Fairness in data-driven methods is very actual topic

- As data-driven systems become more pervasive
- They have the potential to significantly affect social groups

Once you deploy an AI model, performance is not enough

- You might have stellar accuracy and efficient inference
- ...And still end up causing all sort of havoc

This is so critical that the topic is about starting to be regulated

- The EU has drafted Ethics Guidelines for Trustworthy AI
- In some fields, in a few years, models that do not comply with specific rules
- ...May be simply forbidden from being deployed

Loading and Preparing the Dataset

We will run an experiment on the "crime" UCI dataset

We will use a pre-processed version made available by our support module:

In [2]:
```
data = util.load_communities_data(data_folder)
data
```

Out[2]:

	communityname	state	fold	pop	race	pct12-21	pct12-29	pct16-24	pct65up	pctUrban	...	pctForeignBorn	pctBornStateR
1008	EastLampetertownship	PA	5	11999	0	0.1203	0.2544	0.1208	0.1302	0.5776	...	0.0288	0.8132
1271	EastProvidencecity	RI	6	50380	0	0.1171	0.2459	0.1159	0.1660	1.0000	...	0.1474	0.6561
1936	Betheltown	CT	9	17541	0	0.1356	0.2507	0.1138	0.0804	0.8514	...	0.0853	0.4878
1601	Crowleycity	LA	8	13983	0	0.1506	0.2587	0.1234	0.1302	0.0000	...	0.0029	0.9314
293	Pawtucketcity	RI	2	72644	0	0.1230	0.2725	0.1276	0.1464	1.0000	...	0.1771	0.6363
...	...	...	...	...	...	...	...	...	...	...	...	...	...
1758	RockyMountcity	NC	8	48997	0	0.1454	0.2653	0.1247	0.1190	1.0000	...	0.0077	0.8138
1822	Amarillocity	TX	9	157615	0	0.1391	0.2660	0.1244	0.1085	1.0000	...	0.0412	0.6651
2207	WestHaventown	CT	10	54021	0	0.1186	0.2772	0.1318	0.1339	1.0000	...	0.0837	0.7031
1081	Humblecity	TX	5	12060	0	0.1545	0.3184	0.1530	0.0719	1.0000	...	0.0638	0.5983
1867	VanBurencity	AR	9	14979	0	0.1539	0.2826	0.1288	0.1078	1.0000	...	0.0210	0.6810

1993 rows × 101 columns

The target is "violentPerPop" (number of violent offenders per 100K people)

Loading and Preparing the Dataset

We start to prepare the data by identifying all numerical attributes

```
In [4]: attributes, target = data.columns[3:-1], data.columns[-1]
        nf = [a for a in attributes if a != 'race'] + [target]
```

- The only categorical input is "race" (0 = primarily white, 1 = primarily black)
- ...And this is also the attribute that we will use to check for discrimination

The we standardize all numeric attributes as usual

```
In [6]: tr_frac = 0.8 # 80% data for training
        tr_sep = int(len(data) * tr_frac)
        tmp = data.iloc[:tr_sep]

        sdata = data.copy()
        sdata[nf] = (sdata[nf] - tmp[nf].mean()) / (tmp[nf].std())

        sdata[attributes] = sdata[attributes].astype(np.float32)
        sdata[target] = sdata[target].astype(np.float32)
```

Loading and Preparing the Dataset

Finally, we separate the training and test set

In [7]:
```
tr = sdata.iloc[:tr_sep]
ts = sdata.iloc[tr_sep:]
tr.describe()
```

Out[7]:

	fold	pop	race	pct12-21	pct12-29	pct16-24	pct65up	pctUrban	medInc
count	1594.000000	1594.000000	1594.000000	1.594000e+03	1.594000e+03	1594.000000	1.594000e+03	1.594000e+03	1.5940
mean	5.515056	0.000000	0.031995	-1.196580e-09	-2.393160e-09	0.000000	-2.393160e-09	1.555554e-08	-3.589
std	2.912637	1.000000	0.176042	1.000000e+00	1.000000e+00	1.000000	1.000000e+00	9.999999e-01	1.0000
min	1.000000	-0.196135	0.000000	-2.175701e+00	-2.922249e+00	-1.572079	-2.139933e+00	-1.562290e+00	-1.545
25%	3.000000	-0.177169	0.000000	-4.967758e-01	-5.304008e-01	-0.460043	-6.478973e-01	-1.562290e+00	-7.508
50%	5.000000	-0.141106	0.000000	-1.909697e-01	-1.486426e-01	-0.253682	-3.945406e-02	6.843710e-01	-2.136
75%	8.000000	-0.045777	0.000000	2.007248e-01	2.358963e-01	0.052345	5.321295e-01	6.843710e-01	5.6754
max	10.000000	32.775719	1.000000	8.726096e+00	6.657856e+00	7.807232	8.473559e+00	6.843710e-01	6.6732

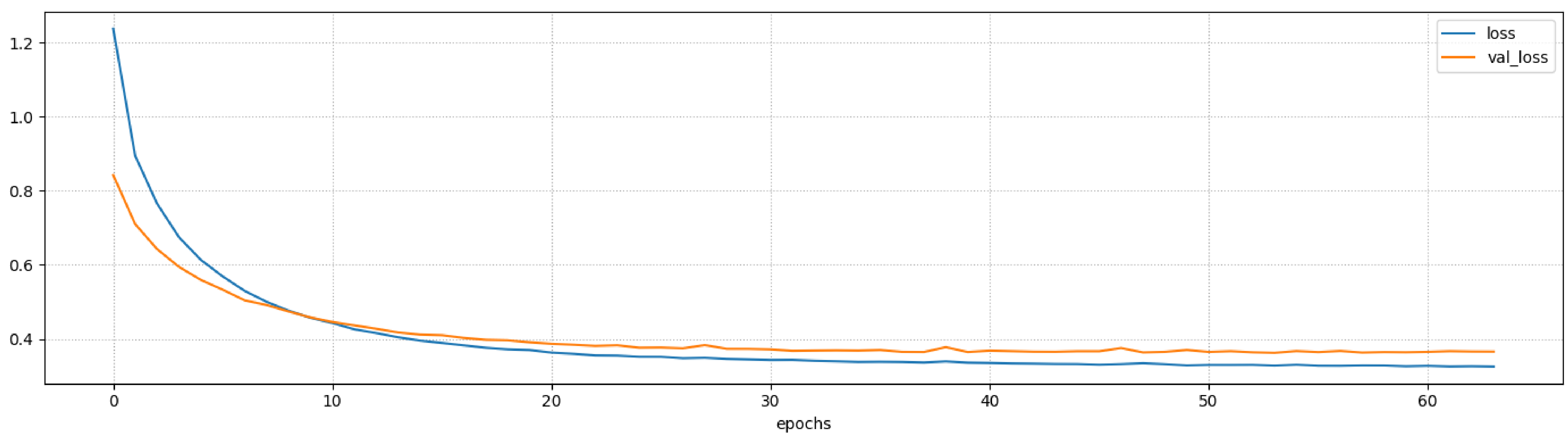
8 rows × 99 columns

Baseline

Let's establish a baseline by tackling the task via Linear Regression

In [8]:
```
nn = util.build_nn_model(input_shape=len(attributes), output_shape=1, hidden=[], output_activati
history = util.train_nn_model(nn, tr[attributes], tr[target], loss='mse', batch_size=32, epochs=
util.plot_training_history(history, figsize=figsize)
```

```
2022-11-28 10:22:32.013117: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFl
ow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following C
PU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```




```
Final loss: 0.3252 (training), 0.3655 (validation)
```

Baseline Evaluation

...And let's check the results

```
In [9]: tr_pred = nn.predict(tr[attributes], verbose=0)
        r2_tr, mae_tr = r2_score(tr[target], tr_pred), mean_absolute_error(tr[target], tr_pred)
        ts_pred = nn.predict(ts[attributes], verbose=0)
        r2_ts, mae_ts = r2_score(ts[target], ts_pred), mean_absolute_error(ts[target], ts_pred)
        print(f'R2 score: {r2_tr:.2f} (training), {r2_ts:.2f} (test)')
        print(f'MAE: {mae_tr:.2f} (training), {mae_ts:.2f} (test)')

R2 score: 0.67 (training), 0.60 (test)
MAE: 0.39 (training), 0.46 (test)
```

- They are definitely not PreCrime level, but they are not bad
- Some improvements (not much) can be obtained with a Deeper model

Linear Regression is an interpretable ML model

- In particular, we can have evaluate the importance of each input attribute
- This can be done in LR by inspecting the weights

We could try this approach to check for discrimination

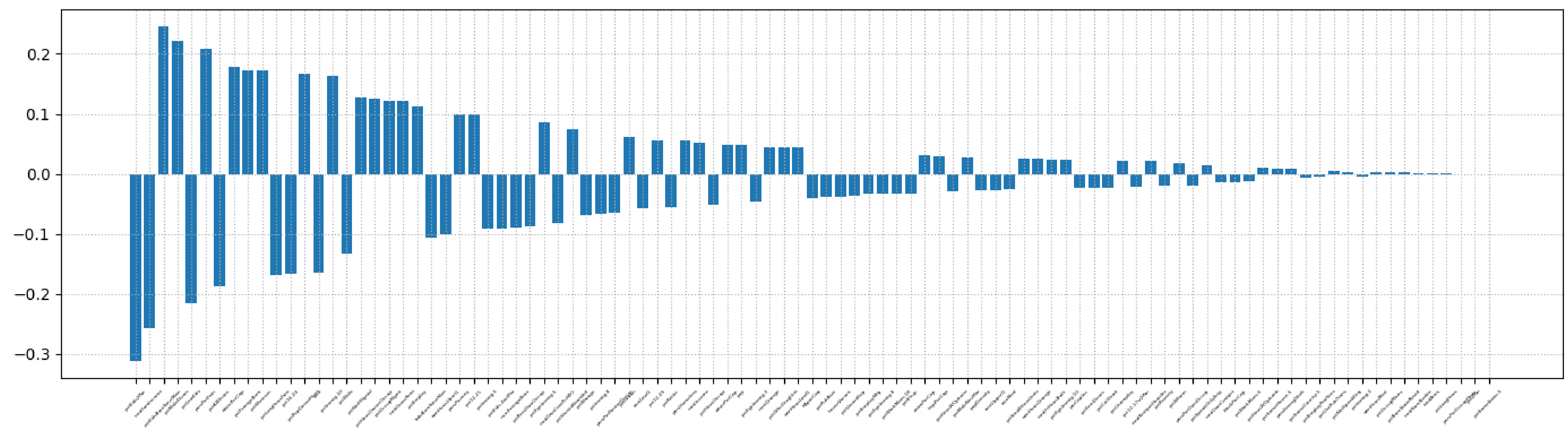
Important Attributes in Linear Regression

Important Attributes in Linear Regression

Let's plot the weights by decreasing (absolute) value

- If all attributes are standardized/normalized (so they have similar ranges)
- ...Then the larger the (absolute) weight, the larger the impact

In [11]:
```
lr_weights = nn.get_weights()[0].ravel()
util.plot_lr_weights(lr_weights, attributes, figsize=figsize)
```

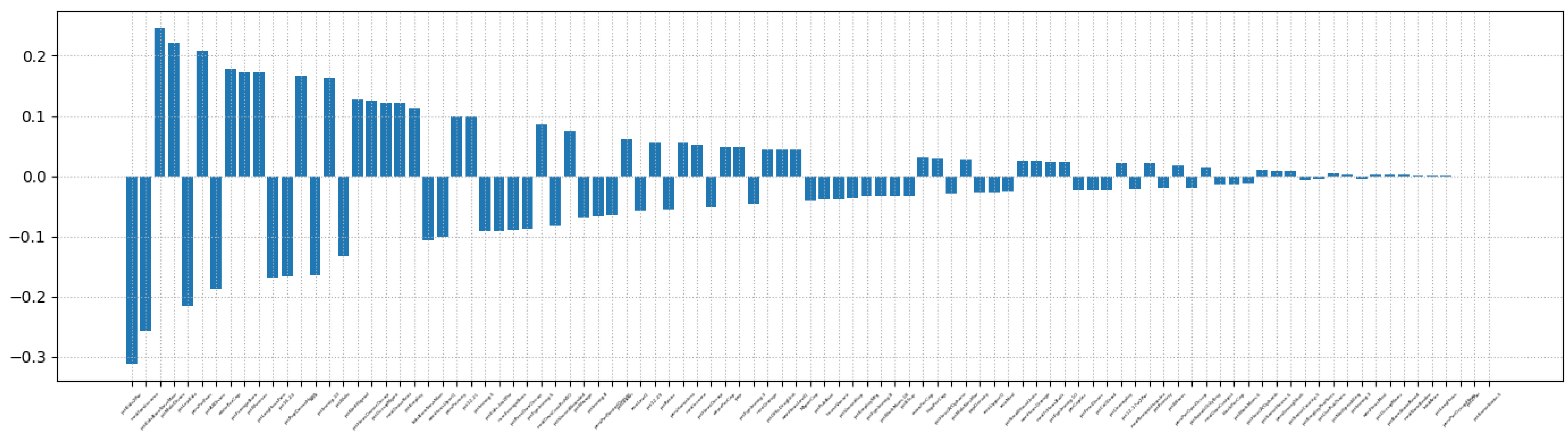


Important Attributes in Linear Regression

Let's plot the weights by decreasing (absolute) value

- If all attributes are standardized/normalized (so they have similar ranges)
- ...Then the larger the (absolute) weight, the larger the impact

In [11]:
```
lr_weights = nn.get_weights()[0].ravel()
util.plot_lr_weights(lr_weights, attributes, figsize=figsize)
```



- Unfortunately, itthere are many large-ish weights

Lasso

We can fix this by adding an L1 regularizer to obtain LASSO (Regression)

The regularizer penalizes weight magnitudes via a fixed rate α, i.e.:

$$f(x, \theta) = \theta^T x + \alpha ||\theta||_1$$

- Attributes for which the loss reduction does not match the regularization rate...
- ...Will be kept at zero, resulting in a sparse weight vector

Lasso is available in scikit-learn, and can be implemented in Keras/Tensorflow

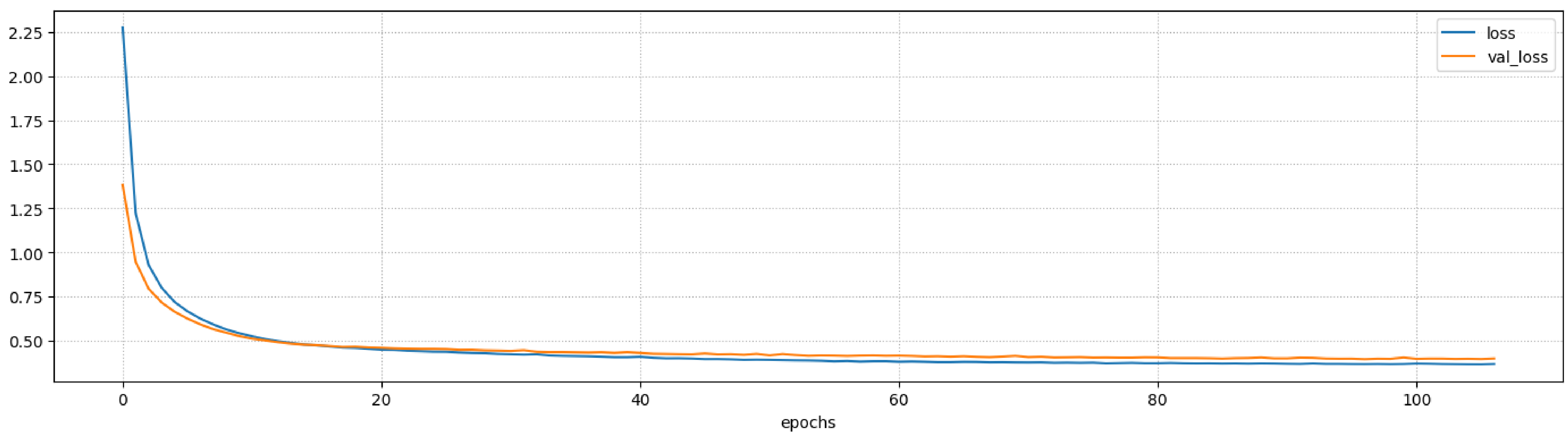
We just need to add L1 regularization over the output neuron:

```
...
model_out = layers.Dense(output_shape, activation=output_activation,
        kernel_regularizer=regularizers.l1(l1=1e-3))(x)
...
```

Lasso

We can train the Lasso model as usual

In [12]:
```
nn2 = util.build_nn_model(input_shape=len(attributes), output_shape=1, hidden=[], output_activat
                          kernel_regularizers=[regularizers.l1(l1=1e-2)])
history = util.train_nn_model(nn2, tr[attributes], tr[target], loss='mse', batch_size=32, epochs
util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.3658 (training), 0.3968 (validation)
```

Lasso Evaluation

The results are on par with Linear Regression

In [13]:
```
tr_pred2 = nn2.predict(tr[attributes], verbose=0)
r2_tr2, mae_tr2 = r2_score(tr[target], tr_pred2), mean_absolute_error(tr[target], tr_pred2)
ts_pred2 = nn2.predict(ts[attributes], verbose=0)
r2_ts2, mae_ts2 = r2_score(ts[target], ts_pred2), mean_absolute_error(ts[target], ts_pred2)

print(f'R2 score: {r2_tr2:.2f} (training), {r2_ts2:.2f} (test)')
print(f'MAE: {mae_tr2:.2f} (training), {mae_ts2:.2f} (test)')
```

```
R2 score: 0.66 (training), 0.61 (test)
MAE: 0.39 (training), 0.45 (test)
```

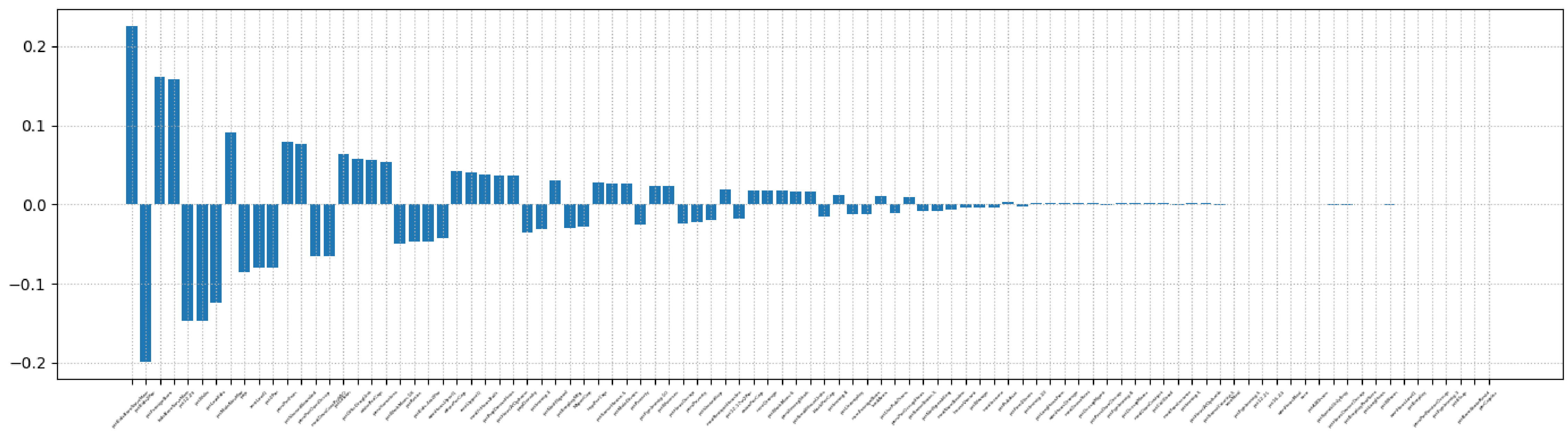
- The L1 term actually acts also as a traditional regularizer...
- ...And may therefore help to prevent overfitting

Important Attributes in Lasso

The main difference between LR and Lasso is in the weight vector

Lasso weights are sparse, i.e. only a few attributes will have a significant impact

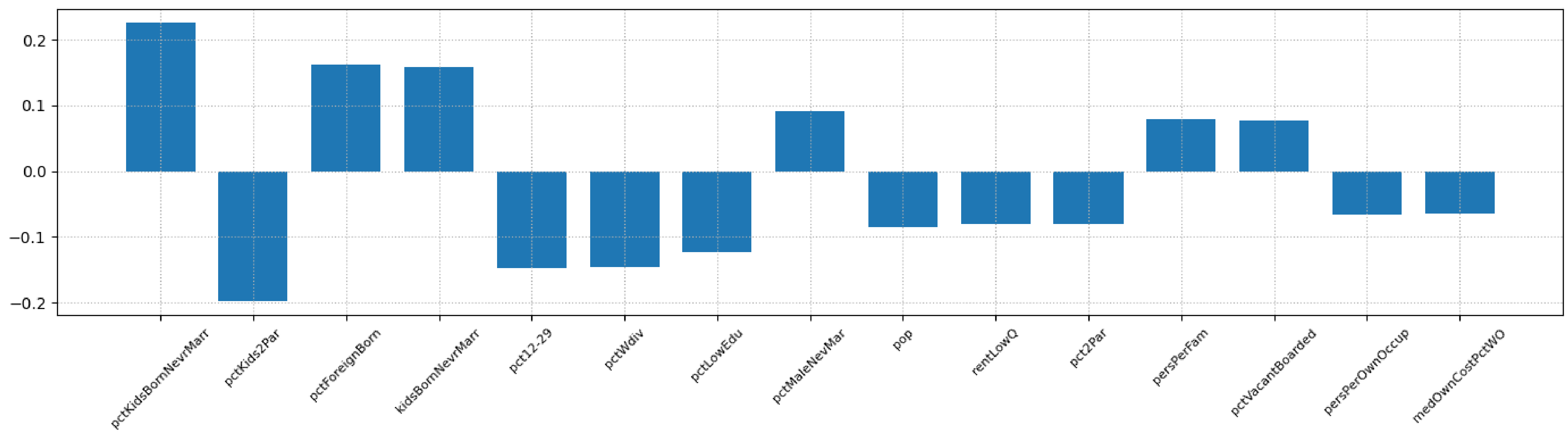
In [14]:
```
lasso_weights = nn2.get_weights()[0].ravel()
util.plot_lr_weights(lasso_weights, attributes, figsize=figsize)
```



Important Attributes in Lasso

Let's zoom in on the 15 most important attributes

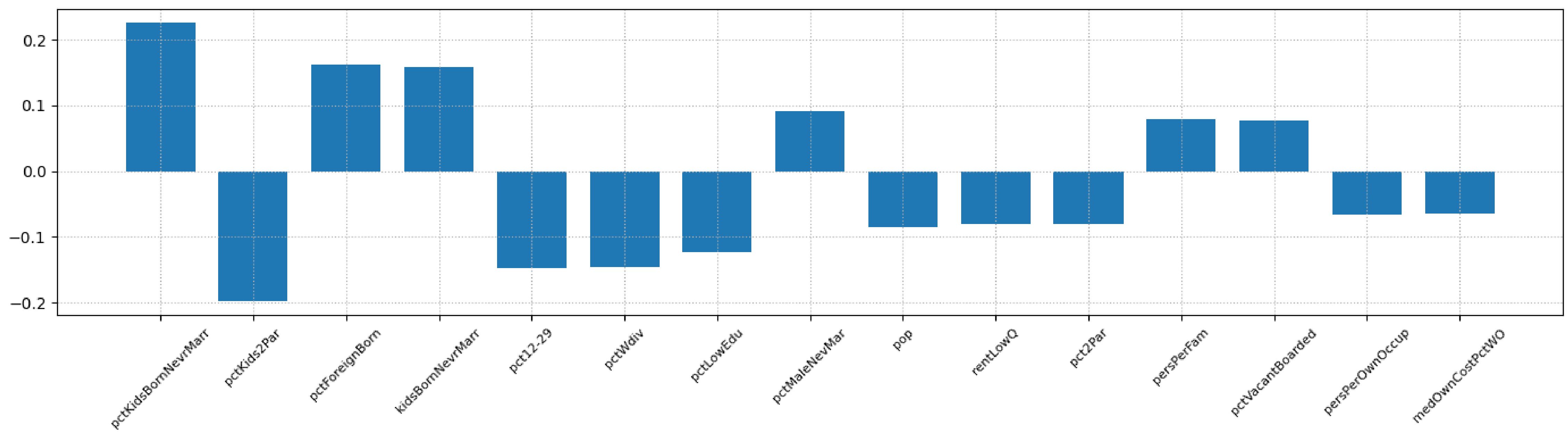
In [16]:
```
lasso_weights = nn2.get_weights()[0].ravel()
util.plot_lr_weights(lasso_weights, attributes, cap_num=15, figsize=figsize)
```



Important Attributes in Lasso

Let's zoom in on the 15 most important attributes

In [16]:
```
lasso_weights = nn2.get_weights()[0].ravel()
util.plot_lr_weights(lasso_weights, attributes, cap_num=15, figsize=figsize)
```



The attribute "race" is nowhere to be seen!

- This is looks reassuring for our potential discrimination concerns
- ...But in fact it is not (and we will proceed to check it)

Fairness Metrics

Fairness Metrics

Measuring fairness is complicated

- As with all things related to metrics, measuring is per-se questionable
- ...But if we want to obtain algorithms, it's a necessary step

Several fairness metrics have been proposed

Here we will focus on the idea of disparate treatment

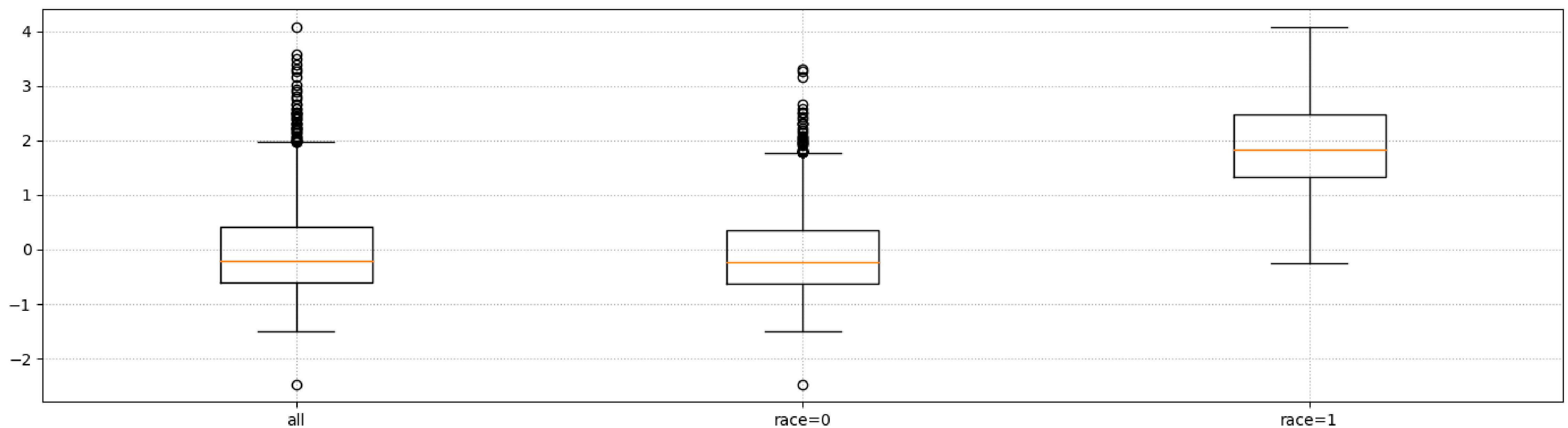
- We will check whether different groups
- ...As defined by the value of a protected attribute ("race" for us)
- Are associated to different predictions

Disparate Treatment

Our model treats the groups differently

...Even if race is not an important attribute

In [17]:
```
protected={'race': (0, 1)}
util.plot_pred_by_protected(tr, tr_pred, protected={'race': (0, 1)}, figsize=figsize)
```

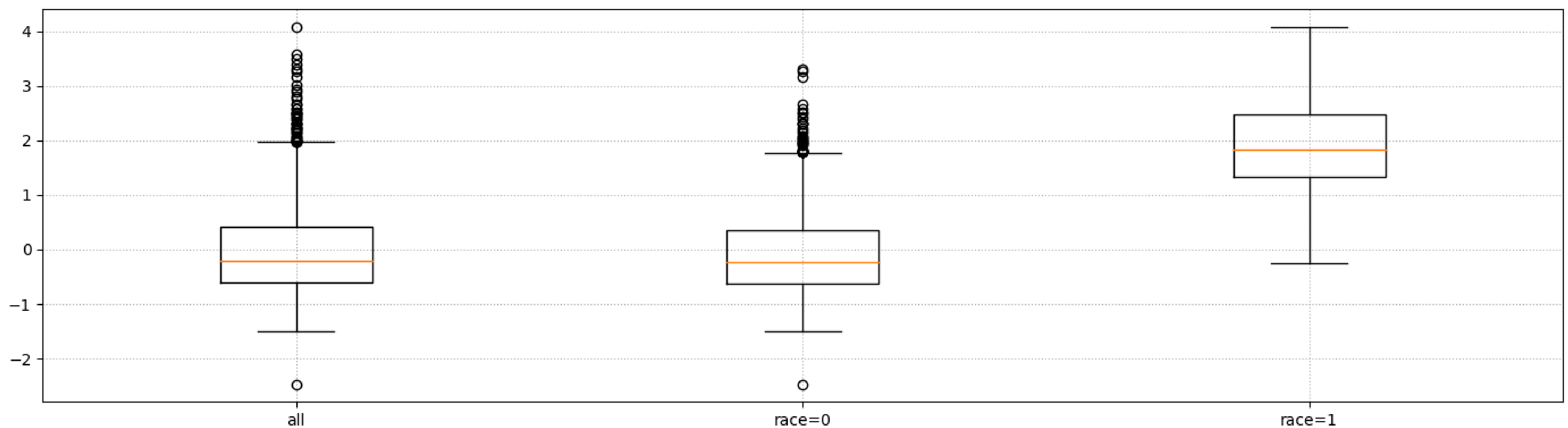


Disparate Treatment

Our model treats the groups differently

...Even if race is not an important attribute

In [17]:
```
protected={'race': (0, 1)}
util.plot_pred_by_protected(tr, tr_pred, protected={'race': (0, 1)}, figsize=figsize)
```



This would happen even if removed the "race" attribute

Discrimination Indexes

Therefore, checking the important attributes is not enough

- We need to measure disparate treatment for the trained model
- ...And as we mentioned there are alternative metrics to do that

We will use the one from thi AAAI paper

- Given a set of categorical protected attribute (indexes) J_p
- ...The Disparate Impact Discrimination Index (for regression) is given by:

$$\text{DIDI}_r = \sum_{j \in J_p} \sum_{v \in D_j} \left| \frac{1}{m} \sum_{i=1}^{m} y_i - \frac{1}{|I_{j,v}|} \sum_{i \in I_{j,v}} y_i \right|$$

- Where D_j is the domain of attribute j
- ...And $I_{j,v}$ is the set of example such that attribute j has value v

DIDI

Let's make some intuitive sense of the DIDI_r formula

$$\sum_{j \in J_p} \sum_{v \in D_j} \left| \frac{1}{m} \sum_{i=1}^{m} y_i - \frac{1}{|I_{j,v}|} \sum_{i \in I_{j,v}} y_i \right|$$

- $\frac{1}{m} \sum_{i=1}^{m} y_i$ is just the average predicted value
- ...For examples where the protected attribute takes specific values
- $\frac{1}{|I_{j,v}|} \sum_{i \in I_{j,v}} y_i$ is the average prediction for a social group

We penalize the group predictions for deviating from the global average

- Obviously this is not necessarily the best definition, but it is something
- In general, different tasks will call for different discrimination indexes

...And don't forget the whole "can we actually measure ethics" issue ;-)

DIDI

We can compute the DIDI via the following function

```python
def DIDI_r(data, pred, protected):
    res, avg = 0, np.mean(pred)
    for aname, dom in protected.items():
        for val in dom:
            mask = (data[aname] == val)
            res += abs(avg - np.mean(pred[mask]))
    return res
```

- `protected` contains the protected attribute names with their domain

For our original Linear Regression model, we get

In [18]:
```python
tr_DIDI = util.DIDI_r(tr, tr_pred, protected)
ts_DIDI = util.DIDI_r(ts, ts_pred, protected)
print(f'DIDI: {tr_DIDI:.2f} (training), {ts_DIDI:.2f} (test)')
```

```
DIDI: 1.94 (training), 2.13 (test)
```

Improving the DIDI

We will try to improve over this baseline

This is not a trivial task:

- Discrimination arises from a form of bias in the training set
- ...And bias is not necessarily bad

In fact, ML works because of bias

I.e. because the training distribution contains information about the test one

- Improving fairness requires to get rid of part of this bias
- ...Which will lead to some loss of accuracy (hopefully not too much)

We will see one method to achieve this result